Streaming-based Processing of Secured XML Documents

Chair for Network- and Data Security
Horst Görtz Institute for IT Security
Ruhr-University Bochum

Juraj Somorovský
Matriculation Number: 108 006 245 025

2.10.2009

Supervisors: Prof. Jörg Schwenk
Dipl.-Inf. Meiko Jensen

Eidesstattliche Erklärung

Ich versichere hiermit, dass ich meine Masterarbeit mit dem Thema

Streaming-based Processing of Secured XML Documents

selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Bochum, den 2. Oktober 2009

Abstract

WS-Security is a standard providing message-level security in Web Services. It allows flexible application of security mechanisms in SOAP messages. Therewith it ensures their integrity, confidentiality and authenticity. However, using sophisticated security algorithms can lead to high memory consumptions and long evaluation times. In the combination with the standard XML DOM processing approach the Web Services servers become a simple target of Denial-of-Service attacks.
This thesis presents a solution for these problems – an external streaming-based WS-Security Gateway. First the difficulties and limitations of the streaming-based approach applied on secured XML documents are discussed. Then the implementation of the WS-Security Gateway is presented. The implementation is capable of processing XML signatures in SOAP messages. The evaluation shows that streaming-based approach enhances the performance and is much more efficient in comparison to standard DOM-based frameworks. It is also proved that the implementation is under certain circumstances resistant against signature wrapping attacks.

Contents

1 Introduction **1**

2 Related work **3**
2.1 XML Signature . . . 3
2.1.1 XML Canonicalization . . . 5
2.2 Web Services . . . 6
2.2.1 Web Services Security . . . 7
2.2.2 Web Services Policy . . . 7
2.2.3 XML Signature in WS Security . . . 8
2.3 Processing XML Documents . . . 10
2.3.1 SAX vs StAX . . . 10
2.3.2 XPath evaluation and SPEX . . . 11
2.4 Threats for Web Services . . . 11
2.4.1 DoS Attacks . . . 11
2.4.2 XML Wrapping Attacks . . . 12
2.5 Event Pipeline Pattern . . . 13
2.5.1 Related Design Patterns . . . 13

3 Concept **15**
3.1 Web Services Security Gateway . . . 15
3.2 Architecture . . . 16
3.3 Handling Security Exceptions . . . 17
3.4 Exclusive XML Canonicalization . . . 18
3.5 Processing XML Signatures in SOAP Messages . . . 19
3.5.1 Problems of Event-based XML Signature Verification . . . 19
3.5.2 SAX-based XML Signature Verification and Reference Handling . . . 21
3.5.3 SAX-based XPath Transformation Handling . . . 23
3.5.4 XML Signature Creation . . . 26
3.6 SOAP Message Serialization . . . 27

4 Implementation **29**
4.1 Package Hierarchy . . . 29
4.1.1 Utilities . . . 30
4.1.2 Signature Handling Utilities . . . 32
4.2 Exclusive XML Canonicalization . . . 34
4.2.1 Utilities . . . 34
4.2.2 ExclusiveSAXCanonicalizer . . . 36
4.2.3 Use of Exclusive Canonicalization . . . 37
4.3 Event Pipeline Pattern Application on WSS Gateway . . . 38
4.3.1 Connection to the SAX Parser . . . 39
4.3.2 SecurityExceptionHandler . . . 40
4.3.3 SOAPMessageHandler . . . 42

4.3.4 SignatureHandler 42
4.3.5 Reference Handlers 43
4.3.6 Serialization Handlers 45

5 Evaluation **49**
5.1 SAX vs DOM 49
5.2 Performance of XPath Referencing 52
5.3 Wrapping Attacks Resistance 55

6 Conclusion and outlook **57**

List of Figures

2.1 Three XML Signature types: Enveloping, Enveloped and Detached 3
2.2 The structure of a SOAP message 6
2.3 SOAP message with XML Signature according to WSS standard 10
2.4 XML Signature Wrapping attack 12
2.5 Streaming-based XML processing 13
2.6 Mediator and Chain of responsibility patterns 14

3.1 Web Services Security Gateway functionality 16
3.2 Web Services Security Gateway and its modules 17
3.3 Handling security exceptions in the WSS Gateway 18
3.4 Event-based XML canonicalization – activity diagram 19
3.5 SOAP message parsing approach with storing SOAP header events 21
3.6 XML Signature verification state diagram 22
3.7 The ReferenceXPathHandler is added dynamically to the pipe 24
3.8 True Match and Might Match in XML documents 24
3.9 State diagram of XPath evaluation 25
3.10 Client side WSS Gateway configuration 26
3.11 State diagram of SOAP message serialization 27
3.12 SOAP message with incorrect XML Signature 28

4.1 The WSS Gateway Package Diagram 30
4.2 The class diagram of the signature handling classes 33
4.3 Classes handling Exclusive XML Canonicalization 35
4.4 Canonicalization utilities classes 35
4.5 Event Pipeline Pattern application on WSS Gateway 38
4.6 SAX Parser connection to the pipeline 39
4.7 Security Exceptions 41
4.8 SecurityExceptionHandler handling the ReferenceMoreResultsException . . . 41
4.9 Processing references in SOAP message 44
4.10 `ReferenceXPathHandler` and its connection with the SPEX engine 45
4.11 Handling XPath transformation with Might match and True match 46
4.12 Serialization Handlers 47
4.13 SOAP message serialization 48

5.1 Evaluation time comparison with maximum 1000 elements 50
5.2 Heap memory consumption comparison with maximum 140,000 elements . . 51
5.3 Evaluation time comparison with maximum 140,000 elements 51
5.4 Evaluation time comparison with maximum 600,000 elements 52
5.5 SOAP messages designed for the XPath evaluation test 53
5.6 Heap memory consumption comparison by the use of XPath referencing . . . 54
5.7 Evaluation time comparison by the use of XPath referencing 54
5.8 SOAP message with two SOAP bodies 55

Listings

2.1 The structure of an XML Signature 4
2.2 The structure of a WSDL file 7
2.3 WS-Policy example 8
2.4 The structure of SOAP message with XML Signature 9

3.1 SOAP message with a binary security token 20

4.1 Example of a signature configuration file 32
4.2 XMLSignatureBuilder.java 34
4.3 AttributesComparator.java 36
4.4 XML Signature Reference element referencing with Id 43
4.5 XML Signature Reference element referencing with XPath 43

1 Introduction

In the past years, an important term in the distributed computing became a Service-oriented architecture (SOA). The SOA provides that the application's (server's) business logic is visible as services to another applications (clients). The nature of the provided services is loosely coupling. The loosely coupling principle makes the services' interfaces independent of their implementations. Therewith the services achieve their most essential properties: reusability and interoperability. These properties gained the SOA quickly much popularity. They can be implemented with many service-based technologies, such as Jini [com07], REST [Fie00] or CORBA [Gro06].
The most common connection technology of SOAs are Web Services [HBN+04]. The Web Services are Web applications that use XML-based SOAP messages [NHK+07] to invoke functions or transport data. The SOAP messages are transferred over internet with standard protocols such as HTTP. As they often include sensitive and confidential data they become a target of many security attacks.
In distributed systems the most common technology to provide security over a network is SSL (Secure Sockets Layer). The SSL creates a communication channel between parties and encrypts all the sent data at the transport-layer. But as the Web Services apply the message-layer architecture, encrypting of the whole SOAP messages is not always adequate. There are also needs to encrypt some concrete data in the SOAP messages (e.g. encrypt usernames, passwords or credit card numbers) that can be decrypted only by the end-party. This allows all the other parties in the network to view other insensitive information. These requirements make the SSL insufficient and lead to the need of a message-level security.
To ensure the message-level security in SOAP messages the Organization for the Advancement of Structured Information Standards (OASIS) defined the WS-Security standard [KL06]. The WS-Security uses two underlying standards: XML Encryption [ER02] and XML Signature [RSH+08]. These standards offer a flexible usage of security mechanisms in SOAP messages and therewith improve the Web Services to provide the integrity, confidentiality and authenticity.
However, the use of security mechanisms in the SOAP messages supports attacks on a Web Services' availability. The Web Services' availability is vulnerable due to the complex security algorithms computations. They need a high memory consumption and processor computation.
The second reason contributing to the performance problem is the standard XML-processing model: DOM [BHH+04]. This model is applied in many platforms. It parses the incoming SOAP message into an object tree. The object tree includes all the XML data. The data can not be evaluated until the whole SOAP message is processed. Again, the storage of all the elements' data causes high memory consumptions and eliminates high performance processing of large SOAP documents [GLH06].
The usage of WS-Security and DOM-based XML processing provoke new kinds of Denial-of-Service attacks. An attacker can for instance send to the Web Service server large SOAP

messages with invalid signatures. According to the DOM, first the whole SOAP messages have to be parsed. Then the signature validation follows. Although the messages are incorrect, the attacker was successful and involved the server in costly computations.
This thesis provides a solution for the problems with evaluation of large XML documents, concretely SOAP messages. It presents a streaming-based Web Services Security Gateway. The Web Services Security Gateway can be used for processing of SOAP messages containing XML signatures. The whole processing (verification and creation) of XML signatures is done with an streaming-based approach. This approach brings two major advantages over the DOM. It parses the incoming messages step-by-step and allows to interrupt the parsing as soon as an invalid element is encountered. This is very important by the processing of SOAP messages with security elements. Additionally the whole document model is not stored in the memory. During the parsing it is allowed to store only the needed elements. Therewith the application's memory consumption can be rapidly reduced.
The next chapters of this thesis are organized as follows. Chapter 2 summarizes related work and related standards. Chapters 3 and 4 contain the concept and implementation of the streaming-based Web Services Security Gateway. Chapter 5 presents the evaluation results. The results and the whole thesis are then concluded in Chapter 6.

2 Related work

2.1 XML Signature

Digital signatures ensure integrity and authenticity of exchanged messages. An XML syntax for digital signatures is defined by W3C and is called XML Signature [RSH+08]. It allows to apply digital signatures to arbitrary parts of XML documents. There are three types of XML Signatures (Figure 2.1):

- Enveloping: The signed element is encapsulated in the XML Signature.
- Enveloped: The signed element encapsulates the XML Signature.
- Detached: The signed element is referenced by URI or xpointer [WJD+02] and can be found in the same document as well as outside of the document.

The structure of an XML Signature is shown in Listing 2.1 ("?" denotes zero or one occurrence; "+" denotes one or more occurrences; and "*" denotes zero or more occurrences). The root element of the XML Signature is the `<Signature>` element, which can have up to four children: SignedInfo, SignatureValue, KeyInfo and Object. The `<SignedInfo>` contains information about the used canonicalization method, signature method and signed elements. Each signed element is referenced through a `<Reference>`. The `<Reference>` carries data about the used digest method and computed digest value. The `<SignatureValue>` element contains the actual value (Base-64 encoded) of the digital signature computed over the `<SignedInfo>`. `<KeyInfo>` and `<Object>` are optional elements. The `<KeyInfo>` carries the public key for signature verification. The `<Object>` is used only in Enveloping XML Signature and contains signed elements.

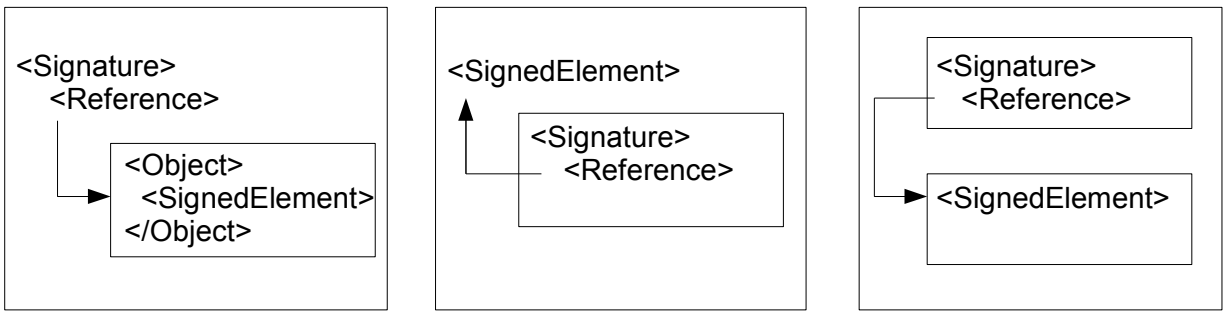

Figure 2.1: Three XML Signature types: Enveloping, Enveloped and Detached

```
<Signature ID?>
   <SignedInfo>
      <CanonicalizationMethod/>
      <SignatureMethod/>
      (<Reference URI? >
         (<Transforms>)?
         <DigestMethod>
         <DigestValue>
      </Reference>)+
   </SignedInfo>
   <SignatureValue>
   (<KeyInfo>)?
  (<Object ID?>)*
</Signature>
```

Listing 2.1: The structure of an XML Signature

Validation of the XML Signature consists of these two phases:

1. Validation of references

   a) Finding elements in the XML document specified by `<Reference>` elements.

   b) Applying the transformations to these elements given in `<Transforms>` (e.g. XML Canonicalization [Boy01], XPath filter [RBH02])

   c) Computing the digest values with appropriate digest methods taken from `<DigestMethod>` elements.

   d) Comparing these results with the values in `<DigestValue>` elements. If one of the computed digest values is different from the appropriate value in the `<DigestValue>`, processing of the XML Signature is interrupted. The signature is invalid, because one of the signed elements has been altered after the signature creation.

2. Signature value validation:

   a) Canonicalization of `<SignedInfo>` (with `<CanonicalizationMethod>`).

   b) Computing the digest value of canonicalized `<SignedInfo>` element.

   c) Applying an asymmetric key algorithm from the `<SignatureMethod>` on the computed hash, then comparing this result with the digest value given in `<SignatureValue>`. If these two values are equal, the XML Signature is valid. Otherwise the signature is invalid and the XML document not authenticated. The reason could be an altered `<SignedInfo>` block or the use of an incorrect private key by the second party, which has created the XML Signature.

The XML Signature creation is very similar to the XML Signature verification. In the first step the digest values of references have to be computed. In the second step the `<SignedInfo>` is canonicalized and the signature value is computed.

2.1.1 XML Canonicalization

Two XML documents containing the same data can differ in their physical representation. They can contain for example different entity structure, character encoding, attribute ordering or number of whitespaces in the element tags. Computing digest values over such XML documents brings different results. Therefore before creating or validating XML Signatures the XML documents (or their signed parts) have to be converted to the canonical form with an appropriate canonicalization method. The XML Canonicalization ensures that semantically identical XML documents give identical digital signatures.
The major changes done by converting to canonical form are listed below:

- The document is encoded in UTF-8
- Line breaks are normalized to `#xA` (no carriage returns, only line feeds)
- Empty element conversion to start-end tag pair
- Normalization of whitespace in start and end tags
- Lexicographic ordering of namespace and attribute axes
- Encoding of special characters as character references

Generally there are two types of XML Canonicalization: Inclusive [Boy01] and Exclusive [RBr02]. Inclusive XML Canonicalization declares all namespaces [BTHL06] used in ancestor elements in the root element of the canonicalized XML fraction. This brings problems, for example by wrapping an XML document with element, that declares another namespace.
It can be considered in the following document, which is also in the canonical form:

```
<n1:elem1 xmlns:n1="http://b.example">
   content
</n1:elem1>
```

This document could be wrapped by another element that declares its own namespaces:

```
<n0:pdu xmlns:n0="http://a.example">
  <n1:elem1 xmlns:n1="http://b.example">
     content
  </n1:elem1>
</n0:pdu>
```

After wrapping and canonicalizing, the canonical form of the element `<elem1>` would be:

```
<n1:elem1 xmlns:n0="http://a.example"
          xmlns:n1="http://b.example">
  content
</n1:elem1>
```

The `<n1>` element uses only the namespace `n1`. However, the Inclusive XML Canonicalization inserts to this element also the namespace `n0` and therewith changes its canonical form. As can be seen, this proceeding could break the XML Signature validation by defining of any new namespace in an arbitrary parent element of the already signed element. Therefore its use is not allowed and must be avoided [MM07]. There are also other problems discussed

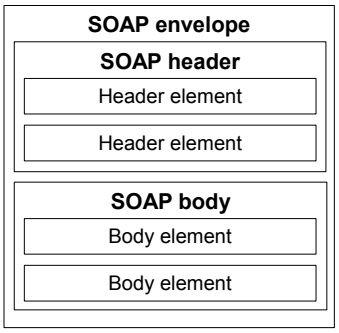

Figure 2.2: The structure of a SOAP message

by W3C [RBr02].
The Exclusive XML Canonicalization was developed to avoid these problems with namespaces declarations. In the Exclusive XML Canonicalization a namespace is declared by an element only if:

1. the element visibly utilizes this namespace (the element or its attribute uses the namespace prefix)

2. and this namespace was not already declared by any ancestor element (that is also canonicalized)

2.2 Web Services

Web Service is a W3C standard [HBN+04] designed to support interoperable interaction over a network between different software applications. In the main, it uses these three technologies:

- XML (EXtensible Markup Language): Language designed to transport and store data.

- SOAP Legacy: Protocol for exchanging structured data between two or more applications [NHK+07]. These data are written into SOAP messages (Figure 2.2).

- WSDL (Web Services Description Language): Language for describing Web Services [MWCR07]. The WSDL document consists basically of service name, binding properties, possible operations (methods that can be executed by Web Service), messages and message elements (Listing 2.2).

SOAP messages are used to communicate with Web Services. They consist mainly of two parts: header (optional element) and body (mandatory element). The header element contains message-specific information, there are made all Security declarations, that are used in the document. The body contains call and response information. The header and body are parts of the envelope element. The envelope element identifies the SOAP message and is always the root element.

```
<definitions name="StockQuote" targetNamespace="http://example.com/
   stockquote.wsdl" ...>
   <types>
      <schema targetNamespace="http://example.com/stockquote.xsd" ...>
         <element name="TradePriceRequest">
            <complexType>
               ...
            </complexType>
         </element>
      </schema>
   </types>
   <message name="GetLastTradePriceInput">
      <part name="body" element="xsd1:TradePriceRequest"/>
   </message>
   <portType name="StockQuotePortType">
      <operation name="GetLastTradePrice">
         ...
      </operation>
   </portType>
   <binding name="StockQuoteSoapBinding" type="tns:StockQuotePortType">
      <soap:binding style="document" transport="http://schemas.xmlsoap.
         org/soap/http"/>
      ...
   </binding>
   <service name="StockQuoteService">
        ...
    </service>
</definitions>
```

Listing 2.2: The structure of a WSDL file

2.2.1 Web Services Security

Web Services Security (WSS) is a standard released by OASIS [KL06]. It enhances SOAP messages to provide common security properties such as message integrity, message authentication and message confidentiality. In doing so, it is built upon other underlying standards such as XML Signature [RSH+08] or XML Encryption [ER02].
WSS discusses three main security mechanisms:

- SOAP message signing
- SOAP message encryption
- Inserting Security tokens (e.g. Kerberos, SAML or X.509 certificate tokens) into SOAP message

2.2.2 Web Services Policy

Web Services Policy [OHV+07] is a framework that allows web services to specify their policy assertions. A policy assertion represents a requirement, capability, or other behavior, for example:

```
<wsp:Policy xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-
   securitypolicy/200702" xmlns:wsp="http://www.w3.org/ns/ws-policy" >
   <wsp:ExactlyOne>
      <wsp:All>
         <sp:SignedParts>
            ...
         </sp:SignedParts>
         <sp:IncludeTimestamp>
            ...
         </sp:IncludeTimestamp>
      </wsp:All>
      <wsp:All>
         <sp:EncryptedParts>
            ...
         </sp:EncryptedParts>
      </wsp:All>
   </wsp:ExactlyOne>
</wsp:Policy>
```

Listing 2.3: WS-Policy example

- encryption and signature algorithms
- usage of security tokens
- association of policies with web services

A special part of this framework is Web Services Security-Policy. Web Services Security-Policy is an OASIS standard [KL07] that indicates the policy assertions which apply to Web Services Security (Protection assertions, Token assertions, Security binding properties, SOAP message security options...).
Policy assertions are grouped into policy alternatives. A set of policy alternatives gives a WS policy. For grouping of policy assertions into WS policy are used two XML tags: `<wsp:All>` and `<wsp:ExactlyOne>`. `<wsp:All>` indicates, that all child node assertions have to be fulfilled. In `<wsp:ExactlyOne>` has to be fulfilled only one policy assertion.
An example of WS-Policy definition gives the XML document in Listing 2.3. This WS-Policy includes two policy alternatives. The first policy alternative declares two policy assertions: signature and timestamp (both policy assertions have to be performed because of the tag `<wsp:All>`).The second policy alternative represents usage of encryption. These two policy alternatives are enclosed in `<wsp:ExactlyOne>` tag, which means, that only one of these alternatives has to be executed.

2.2.3 XML Signature in WS Security

Inserting XML Signature into SOAP message was first described by W3C's standard [BFH+01]. Web Services Security and Basic Security Profile extend this standard and place some constraints on the use of XML Signatures to practice the highest security level. The main constraints are:

- Use of Detached Signature: Enveloping signatures encapsulate signed XML data and

therefore their use is inappropriate – they must not be used. Enveloped signatures should also be avoided, because they limit the ability to process data. On the other hand, detached signatures allow easy referencing of any data in the SOAP body and therefore are encouraged.

- Use of Exclusive XML Canonicalization: Inclusive XML Canonicalization must not be used because of the problems described in Section 2.1.

- Use of specific algorithms: Some algorithms must (or should) be avoided due to their known security weakness, e.g. MD4 [Riv90], MD5 [Riv92].

XML Signature is placed into the header of a SOAP message. An example of a typical SOAP message including XML Signature according to WS Security standard is below in Listing 2.4:

```
<env:Envelope xmlns:env="...">
  <env:Header>
    <wsse:Security xmlns:wsse="..." xmlns:ds="..." xmlns:wsu="...">
      <wsu:Timestamp wsu:Id="Timestamp">
        <wsu:Created>2009-08-10T16:36:51.984Z</wsu:Created>
        <wsu:Expires>2009-08-11T16:36:51.984Z</wsu:Expires>
      </wsu:Timestamp>
      <ds:Signature Id="Signature">
        <ds:SignedInfo>
          <ds:CanonicalizationMethod Algorithm="..."/>
          <ds:SignatureMethod Algorithm="..."/>
          <ds:Reference URI="#Timestamp">
            ...
          </ds:Reference>
          <ds:Reference URI="#body-element">
            ...
          </ds:Reference>
        </ds:SignedInfo>
        <ds:SignatureValue>gH74u...</ds:SignatureValue>
      </ds:Signature>
    </wsse:Security>
  </env:Header>
  <env:Body>
    <Signed-Element wsu:Id="body-element" />
  </env:Body>
</env:Envelope>
```

Listing 2.4: The structure of SOAP message with XML Signature

The structure of this document is also depicted in Figure 2.3. The `<Signature>` element in this SOAP document contains two references. One is pointing to the `<Timestamp>` element in header, the second is pointing to an element in body. This is a typical scenario where the first reference ensures the freshness of the message and the second reference ensures message integrity and authenticity.

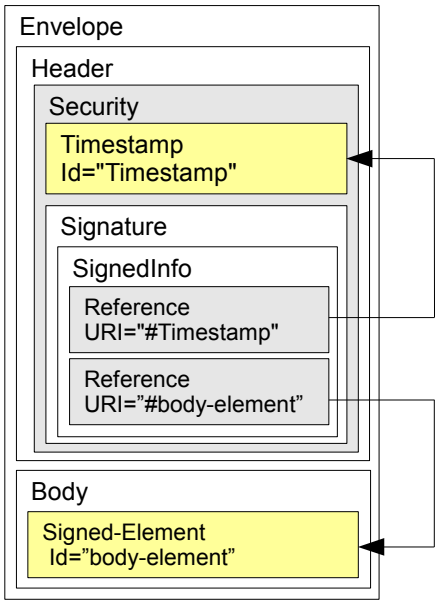

Figure 2.3: SOAP message with XML Signature according to WSS standard

2.3 Processing XML Documents

Two different programming approaches exist for working with XML documents: tree-based and streaming-based (also called event-based). An example of tree-based approach gives DOM [BHH+04], which is specified by W3C. DOM provides maximum flexibility for developers as they can dynamically access and change every node in a DOM tree. But before doing this the whole document has to be read and stored in the memory. This leads to a big disadvantage of DOM – high need for memory consumption during the processing of large XML documents.
Streaming-based paradigms such as SAX [Meg04] or StAX [Mic06c] process XML documents in real time. They do not store every element as an object in the memory and can generate output immediately or interrupt parsing. This brings much more efficiency and is considered preferable for high performance applications. On the other hand, each document is processed only once and there is no possibility to go back during the parsing. Therefore, this paradigm needs much more skilled applications' architectures. Their design is discussed later in this chapter.

2.3.1 SAX vs StAX

These two most used streaming-based approaches have one major difference: SAX uses *push* parsing model, StAX uses *pull* parsing model.
By pull parsing a client application calls the methods of the parser when it needs to interact with an XML document. It controls the flow of a thread and pulls information only if it is needed. By push parsing the thread flow is controlled by the parser. It pushes the information about every element node to the client application, until the document ends.
Pull parsing brings some benefits in comparison to push parsing. A client can write new

XML elements, control the thread flow or read multiple documents at one time in single thread. By contrast, with push processing, the client needs only accept and process events from the parser.

2.3.2 XPath evaluation and SPEX

Querying large XML documents with a SAX approach is a problem discussed in the last years. There have been proposed many different solutions. One of these solutions gives SPEX [BCD+05]. SPEX is a project developed at the University of Munich and stands for Streamed and Progressive Evaluation of XPath queries. It is based on the SAX interface events and for the evaluation doesn't have to create any XML DOM tree. The major benefit is that it delivers the result of the XPath expression [DC99] on the fly as soon as possible. XPath provides evaluation of backward- and forward-navigation expressions. The backward navigation needs to store already parsed stream data and it can be efficiently evaluated only by DOM processing. Therefore, the SPEX project includes the XPath Rewriter. It analyzes all the XPath expression's parts and if it finds any backward navigation, it replaces it with a forward navigation expression. This way is for instance rewritten the expression `/descendant::node()/child::A[parent::B]` into `/descendant::B/child::A`.

2.4 Threats for Web Services

In recent years Web Services became used in many applications. Service Oriented Architecture provides users (or other applications) with methods for accessing databases or processing secret and reliable data. Therefore, Web Services became also a field of interest for many attackers. In this section are mentioned two important attacks on Web Services: DoS attacks and Wrapping attacks (attacks working on other protocol layers or using other Web Services vulnerabilities, like sql injection attacks, are beyond the scope of this thesis).

2.4.1 DoS Attacks

Denial of service attacks on Web Services try to make Web Service Servers unavailable. These attacks are much more easier, when Web Service servers use DOM. The DOM approach parses always the complete message and builds an XML tree structure. The SOAP message could not be processed earlier than the whole document parsing ends. This property supports some kinds of DoS attacks [Lin04]:

- Recursive Payloads: An attacker tries to break a Web Service server sending XML documents that are very deep nested (e.g. more than 100,000 elements).
- Oversize Payloads: Very similar attack that consists of creating large SOAP messages with many elements.

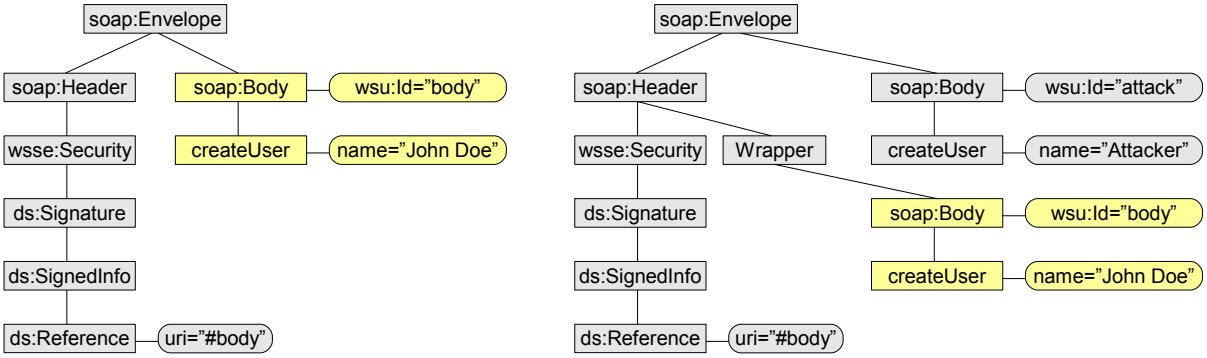

Figure 2.4: XML Signature Wrapping attack: Original (left) and Wrapped (right) SOAP message

- Modified SOAP message with an XML Signature: Verification of XML Signatures takes too much effort. An attacker can engage a server into costly signature verification sending SOAP messages with changed signature values.
- Encrypted large blocks: Recursive and Oversize Payload attacks could be detected with a message schema validation. But it is not possible to do a schema validation, when the message is encrypted. Therefore, sending SOAP messages with encrypted large blocks forces the server into long computations.

2.4.2 XML Wrapping Attacks

In 2005, there was presented a new kind of attack on XML Signatures – XML Wrapping attack [MA05]. This attack exploits the free structure of XML documents and flexible referencing of signed elements in XML Signatures so that an attacker can send a valid SOAP message with altered data. For example, an attacker can replace the whole SOAP body while the SOAP message signature is still considered as valid.
SOAP message processing consists mainly of two parts: validating of XML Signatures (and other security elements handling) and executing instructions in SOAP body. These two parts use different element accessing approaches. XML Signature searches for elements according to their Ids and Web Server application processes commands found in SOAP body. As these two parts are treated independently, this gives attackers a possibility to perform the XML Signature Wrapping attacks.
For instance, it can be considered a SOAP message illustrated in Figure 2.4 (left). This SOAP message has been sent by an administrator and creates a new user – John Doe. When an attacker eavesdrops the communication channel and gets this message, he could easily exchange message body. By Rewriting this message to the form depicted in Figure 2.4 (right) achieves an attacker completely new functionality – creating a user named Attacker. But this attacker's proceeding can not be noticed by the Web Service Server. XML Signature is namely valid because the body with the Id `body` is left unchanged.

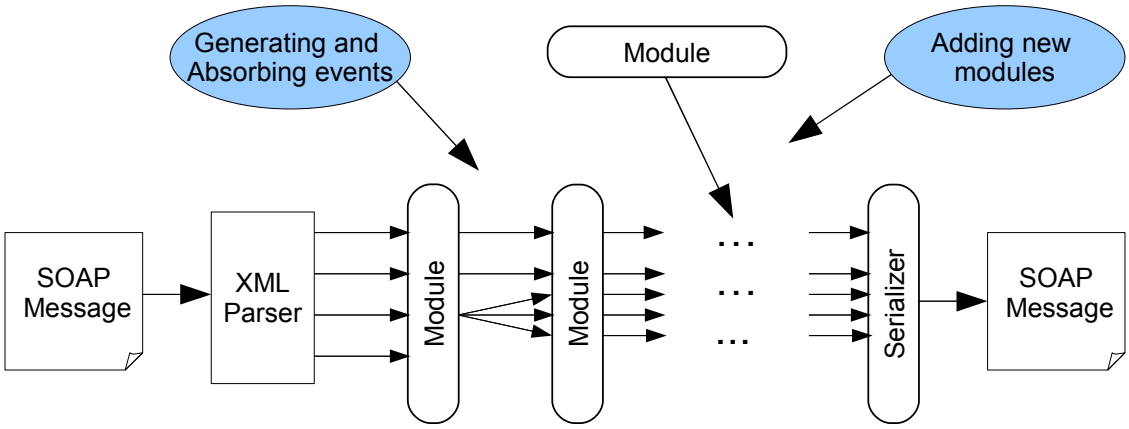

Figure 2.5: Streaming-based XML processing

2.5 Event Pipeline Pattern

Common usage of streaming-based XML parsers can be found in pipelined XML processors. The pipelined XML processor is a system of modules, which together create a pipe. The first module communicates with streaming-based XML parser and expects generated events. When the event comes, it is passed to the next modules in the pipe. This way, each module can get and process generated events. An example of such document processing gives the Event pipeline pattern [JG07].
The application that uses Event Pipeline pattern consists usually of three parts: a document parser, some filtering or checking modules and a sink (e.g. serializer, which buffers a message). If the structure of the XML document is known, modules can be added statically to the pipe on the beginning of document parsing. Otherwise, there is also an opportunity to insert modules at runtime during the parsing. When a module is added to the pipe, it automatically gets all the events passed or generated by its ancestors in the pipe. The implementation logic in the module tells, what to do with the event: It can be processed, absorbed, passed to the next module or it can be generated a completely new event (Figure 2.5).

2.5.1 Related Design Patterns

There are two design patterns, developed by the Gang of Four [FF04] [EGV95], that have got similar properties to the Event Pipeline Pattern: Mediator and Chain of responsibility (Figure 2.6).
The Mediator pattern uses star topology. In the middle stands a mediator object, which controls the flow of all events. If an event comes, it is distributed to all objects, that need it. This way the mediator pattern upholds reusability and decouples the objects from system. Centralized logic also simplifies maintenance. Then again, with adding new modules the implementation becomes very complex. It is as well needed implementation update for every change in event distribution logic. These are the major drawbacks in comparison to Event Pipeline pattern, where the modules can be developed independently and the final

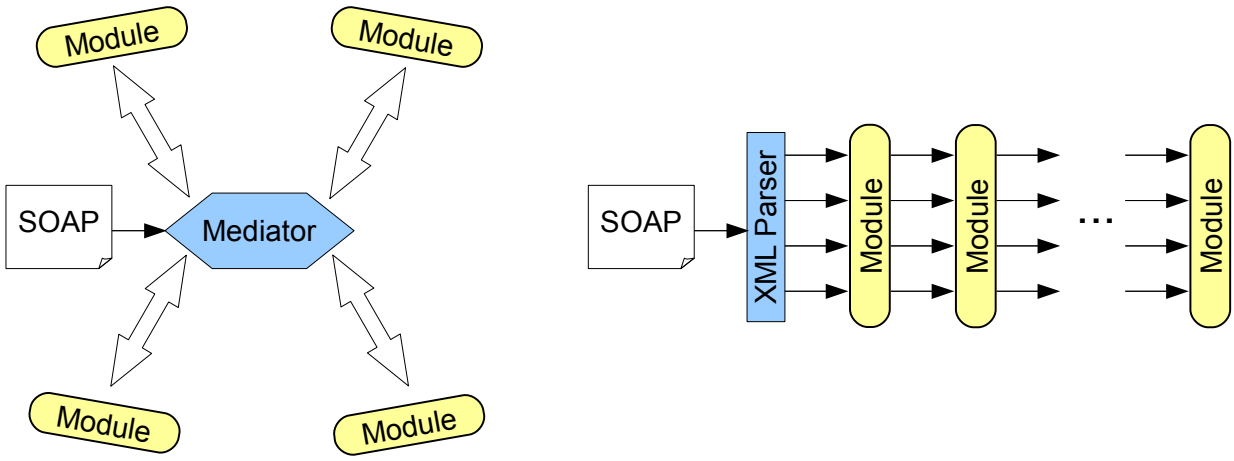

Figure 2.6: Mediator and Chain of responsibility patterns

application is more transparent.
Design and functionality of the Chain of Responsibility pattern is very similar to Event Pipeline pattern. Both are handling events that flow through a pipe. The major difference lies in handling incoming events. If a module in the Chain of Responsibility pattern gets an event, it can process it or send to a next module. The Event pipeline module has got more independent options: process event, send event to a next module, generate new event or completely transform this event. This difference gives a developer more flexibility.

3 Concept

In the previous chapter there were discussed attacks on SOAP messages and expensive methods for their parsing. This thesis presents a solution for these problems, an Event-based **We**b **S**ervices **Se**curity **Ga**teway (WeSSeGa). Its concept and basic architecture are shown in this section.

3.1 Web Services Security Gateway

The implementation of the Web Services Security Gateway adopts an event-based approach for SOAP messages processing. In contrast to DOM, event-based processing has got many advantages: it is memory efficient, faster and can stop XML document handling before the whole document is read. These advantages can be employed by Web Services Security handling.
The WSS Gateway is intended to be deployed on the external server between a Web Service peer (client or server) and a network. This allows to add security capabilities to peers that do not implement any WS-Security standards. A Request-Response scenario with SOAP message including XML Signature looks then as follows (Figure 3.1):

1. WS Client creates a new SOAP message. The message is sent to Client side WSS Gateway.

2. Client side WSS Gateway takes the incoming message and adds XML Signature. SOAP message with new constructed WS-Security elements is forwarded through open network to the WS Server.

3. Server side WSS Gateway gets the sent SOAP message and verifies it. If it is valid, it is forwarded to the WS Server. Otherwise, it is thrown away.

4. WS Server gets a SOAP message without any WS-Security elements so he can concentrate on SOAP body processing.

5. After processing the SOAP message, the response is sent the same way as the request, to WS Client.

This scenario shows that there have to be implemented two sides of WSS Gateway. Each side is handling incoming SOAP messages other way. Server side WSS Gateway is the more important part. It is responsible for verification of WS-Security. This must be done fast and without large memory consumption. Client side WSS Gateway complements the Server side

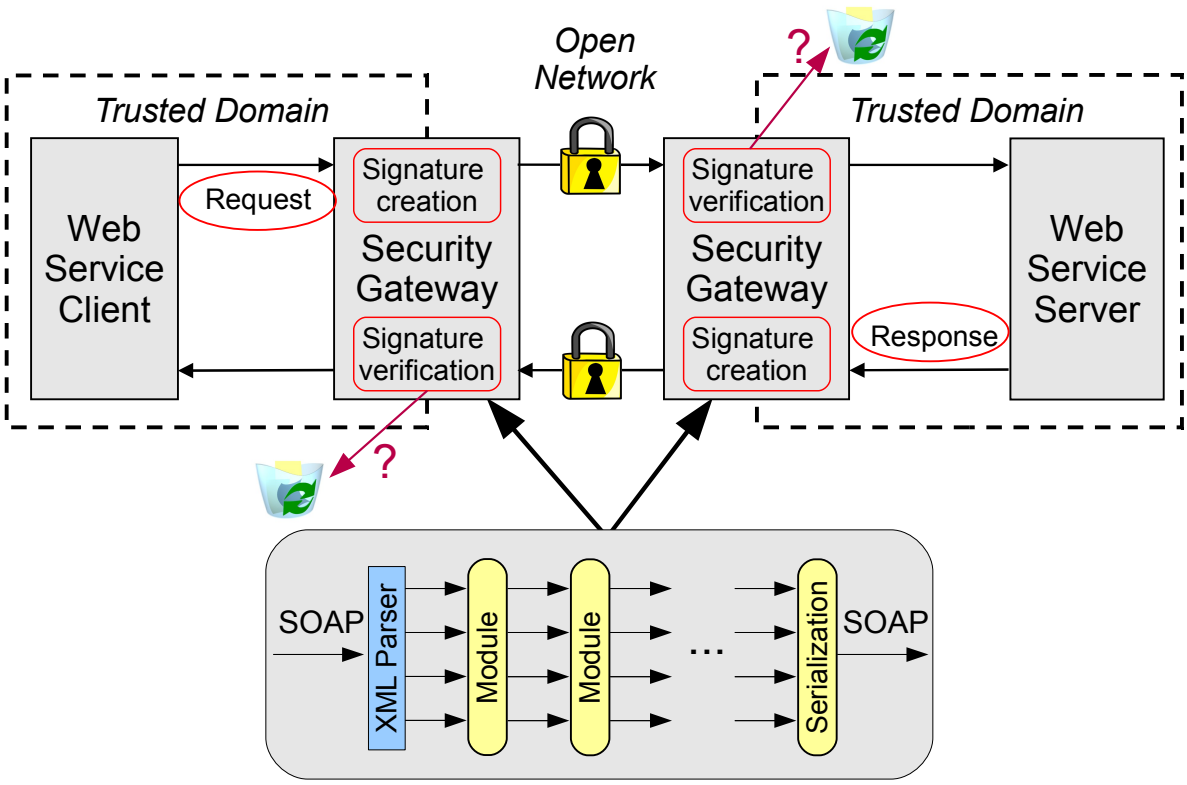

Figure 3.1: Web Services Security Gateway functionality

to have a complete WSS Gateway pair aware of secure SOAP message exchange.
As the WS-Security standard is too widespread, this thesis concentrates on the XML Signature handling in WS-Security.

3.2 Architecture

The architecture of WSS Gateway is based on the Event Pipeline Pattern discussed in previous section. Thus the WSS Gateway consists of an event-based parser and more modules (Figure 3.2). When a SOAP message comes, it is handled by an event-based parser. The parser generates events and pushes them to the pipe. Each module in the pipe is responsible for handling incoming events: it can compute some operation, send it forward, absorb it or generate completely new event. Therefore the sequence of modules is very important.
It was decided to use these event handlers in WSS Gateway:

1. `SecurityExceptionHandler` handles security exceptions coming back from all other event handlers.

2. `SOAPMessageHandler` controls the structure of SOAP messages and generates four new events for the followers in the pipe: `startSOAPHeader`, `endSOAPHeader`, `startSOAPBody`, `endSOAPBody`.

3. `ReferenceXPathHandler` handles XPath transformations and computes digest values

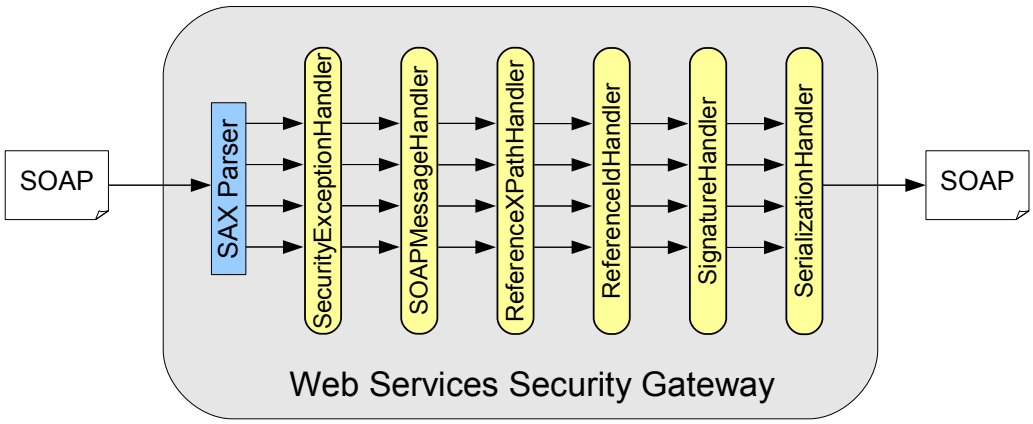

Figure 3.2: Web Services Security Gateway and its modules

of the referenced elements.

4. `ReferenceIdHandler` handles elements referenced with Id and computes their digest values.

5. `SignatureHandler` processes XML Signature verification.

6. `SerializationHandler` is the last module in the pipe and is responsible for serialization of incoming events and reconstruction of SOAP messages. Its output is directed to one of the Web Service peers.

For the implementation of WSS Gateway there have been considered two event-based java APIs that were introduced in the previous chapter: SAX and StAX. Both models are designed for Pipelined XML processing. StAX is preferred in many applications. It offers pull parsing, handling more documents at once, writing new elements and jumping in the file. But for the Event-based WSS Gateway there are no needs for doing this. SOAP document has to be parsed from begin to end and handling multiple documents can be implemented using threads. The last handler, Serializer, looks after serialization and writing of new document. These facts argue for simpler SAX parser that is used in WS-Security Gateway.

3.3 Handling Security Exceptions

The events generated by SAX Parser go through the pipe in sequence from the first until the last module. When the module gets an event, it processes it accordingly its functionality. Handling an event can cause a security exception (e.g. by reference verification). The thrown security exception is directed back to the SAX parser and can interrupt its parsing.
To handle security exceptions thrown by pipeline's modules the `SecurityExceptionHandler` has been created. This handler must be inserted on the first place in the pipe so it can handle all the exceptions thrown by next modules. When a security exception comes, it can decide, what to do with it:

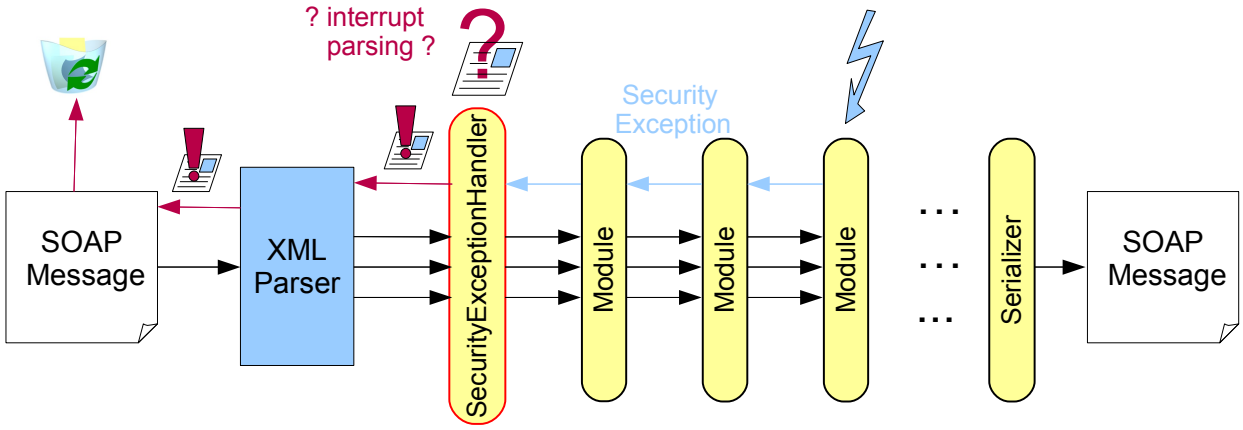

Figure 3.3: Handling security exceptions in the WSS Gateway

- Ignore it.
- Log a warning.
- Interrupt XML parsing by passing the exception to the SAX parser (the default behavior).

An example of the data flow after throwing a `SecurityException` is depicted in Figure 3.3: One of the modules in the pipe throws a `SecurityException`. The `SecurityException` flows back to the `SecurityExceptionHandler`. Its task is to decide, if the parsing has to be interrupted. If yes, the `SecurityException` is brought to the SAX Parser, which stops the parsing and throws the SOAP message away. Otherwise the document parsing continues.
This approach gives an overview in handling all the exceptions that can come from processed SOAP messages: exceptions thrown by signature verification, reference verification or bad algorithm usage. The whole exception handling is defined in one class and therefore the implementation becomes clear and simple.

3.4 Exclusive XML Canonicalization

XML canonicalization is used by every signature and reference digest computation. It needs to solve namespaces declaration problems, attributes sorting, document encoding or replacement of special characters. Execution of all these tasks makes a transformation of XML parts into canonical form one of the bottlenecks of each application. As the WSS Gateway processes mainly XML signatures, developing a fast and lightweight XML canonicalization is the most important goal.
WSS Gateway offers the implementation of one XML Canonicalization type – Exclusive XML Canonicalization. Its functionality is inserted in one independent module. This module is used in three separate handlers. `ReferenceXPathHandler` and `ReferenceIdHandler` use XML Canonicalization for computing of references' digest values. `SignatureHandler` needs it to compute the digest of the SignedInfo block.
The canonicalization module implements three basic SAX events: `startElement`, `charac-`

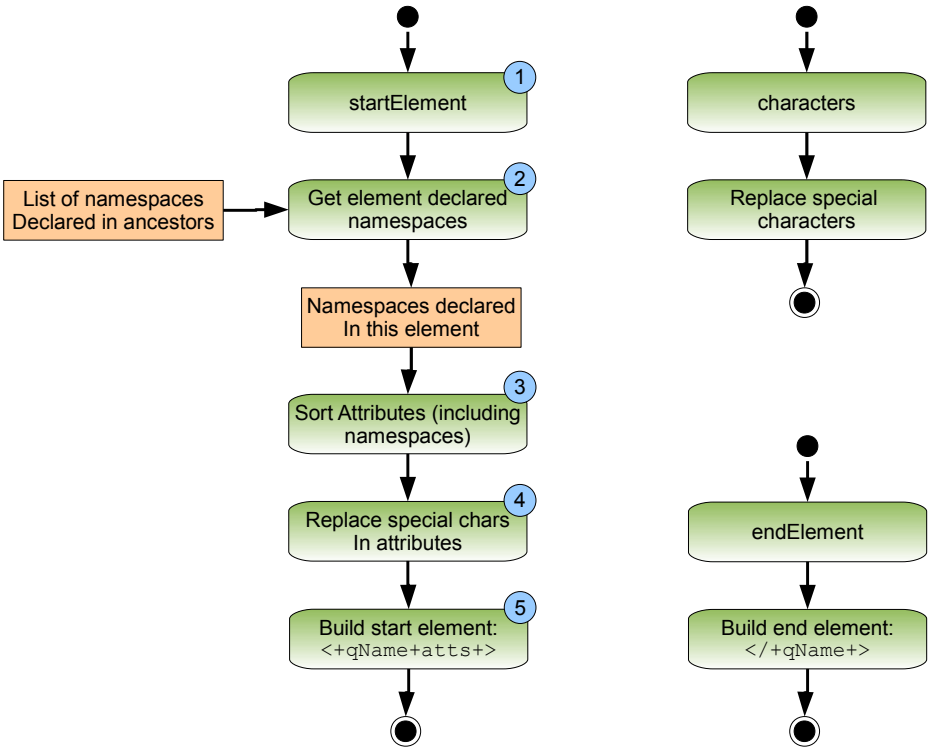

Figure 3.4: Event-based XML canonicalization – activity diagram

`ters` and `endElement`. Handling of these SAX events is described in the activity diagram in Figure (3.4). As it can be seen in this diagram, the most difficult part of XML Canonicalization is processing start element tags: When a `startElement` event comes (1), the canonicalization module takes the already declared namespaces (2). These namespaces are stored in a list. Each namespace declaration includes, along with the namespace prefix and URI, also a level of the declaration in the XML tree. The declaration level serves for the namespaces administration (when the parser later leaves this level, the namespace can be removed from the list). The canonicalization module chooses then only those namespaces that are visibly utilized by the current element and are not included in the list of already declared namespaces. The result is then sent to the next step (3). There are sorted all the attributes including filtered namespaces. At the end special characters are replaced (4) and the element tag is built (5). Processing `characters` and `endElement` events is simple and is clearly shown in the diagram.

3.5 Processing XML Signatures in SOAP Messages

3.5.1 Problems of Event-based XML Signature Verification

XML Signature verification begins with the start of `<Signature>` element. This element is defined in SOAP header. When Signature appears, its properties are saved and the signature value is verified. If it is valid, there have to be validated digest values of referenced elements.

```
<env:Envelope xmlns:env="...">
   <env:Header>
      <wsse:Security xmlns:wsse="..." xmlns:ds="..." xmlns:wsu="...">
         <wsse:BinarySecurityToken wsu:Id="reqToken" ValueType="...#
            X509v3">DFIEZzCCA9CgAwIBAgIQEmtJZc0...</
            wsse:BinarySecurityToken>
         <ds:Signature Id="Signature">
            <ds:SignedInfo>
               <ds:CanonicalizationMethod Algorithm="..."/>
               <ds:SignatureMethod Algorithm="..."/>
               <ds:Reference URI="#reqToken">
                  ...
               </ds:Reference>
            </ds:SignedInfo>
            <ds:SignatureValue>gH74u...</ds:SignatureValue>
         </ds:Signature>
      </wsse:Security>
   </env:Header>
   <env:Body>
      ...
   </env:Body>
</env:Envelope>
```

Listing 3.1: SOAP message with a binary security token

These elements can be referenced with Ids and can be transformed with XPath Filter 2.0 [RBH02] (other reference methods and transformations are out of scope of this thesis).
In Event-based XML processing, there is no problem to find an element referenced with forward references. The problem comes, when the reference refers to an already parsed element (Listing 3.1). There are many approaches handling this issue:

- All the elements' digests are computed: this leads to a high memory consumption and could bring worse results than DOM parsing.
- It is constrained that only Id references are used and all the elements containing attribute Id are computed [GLH06]: with specific attacks, this could lead to same problems as in the first approach.
- Only signing references in body is allowed [LCSG05]: First there are read the XML Signature properties in the header of SOAP message. Later in the body only the needed digests are computed. This approach is very clean and memory saving, but doesn't allow to sign security elements (timestamps...) in the SOAP header.

This thesis concentrates on the handling of SOAP messages. SOAP messages contain all security definitions in the header and business logic in the body of the document. Therefore it has been decided to divide XML Signature processing in two sections: header and body processing. Generally the header contains not so many elements and enables storing of its data without a high memory demand. This consideration leads to two approaches for SOAP message parsing:

1. Storing the header events: During the SOAP header parsing, all the events are stored temporarily in the first event handler (Figure 3.5). This handler processes WS-Security and stores also all the relevant data for XML Signature verification. When the header

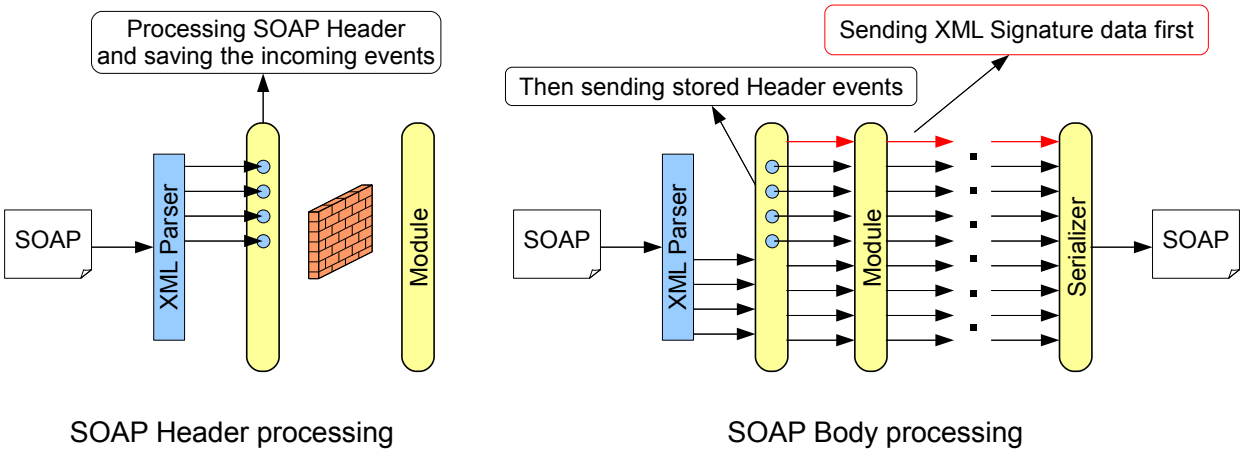

Figure 3.5: SOAP message parsing approach with storing SOAP header events

ends, the information about needed references are sent to other modules. Afterward all the halted events are pushed again to the pipe. This way, the handler processing references gets the data about signed elements always before the signed elements come – also if they are in the SOAP header.

2. Computing digests of all header elements with an attribute Id: In the SOAP header there are computed digests of all elements that could be referenced – it means elements containing an Id attribute. In the SOAP body are then computed only digests of needed references.

Both approaches allow to sign all the elements in the SOAP message. This is an advantage over the techniques in other event-based gateways. For the implementation of WSS Gateway, it has been decided to use the second approach, which is investigated in the next section.
As an extension of this processing, it is allowed to make an XPath Filter 2.0 transformation on the body of SOAP document. This XPath transformation is very simple: it starts evaluating only direct in the body element (which is referenced with an XPointer) and executes only one intersect function. Although, this approach could be considered as a good security extension against wrapping attacks.

3.5.2 SAX-based XML Signature Verification and Reference Handling

`SOAPMessageHandler` generates four new events: `startSOAPHeader`, `endSOAPHeader`, `startSOAPBody`, `endSOAPBody`. With these events is SOAP message parsing divided into five basic states, which makes the signature processing easier. The states and their adequate proceeding is depicted in Figure 3.6:

- `SOAP_INITIAL_STATE`: It is only an initial state between the start of SOAP document and the start of SOAP header.
- `SOAP_HEADER_RUNNING`: This state is splitted into two parallel proceedings. On the

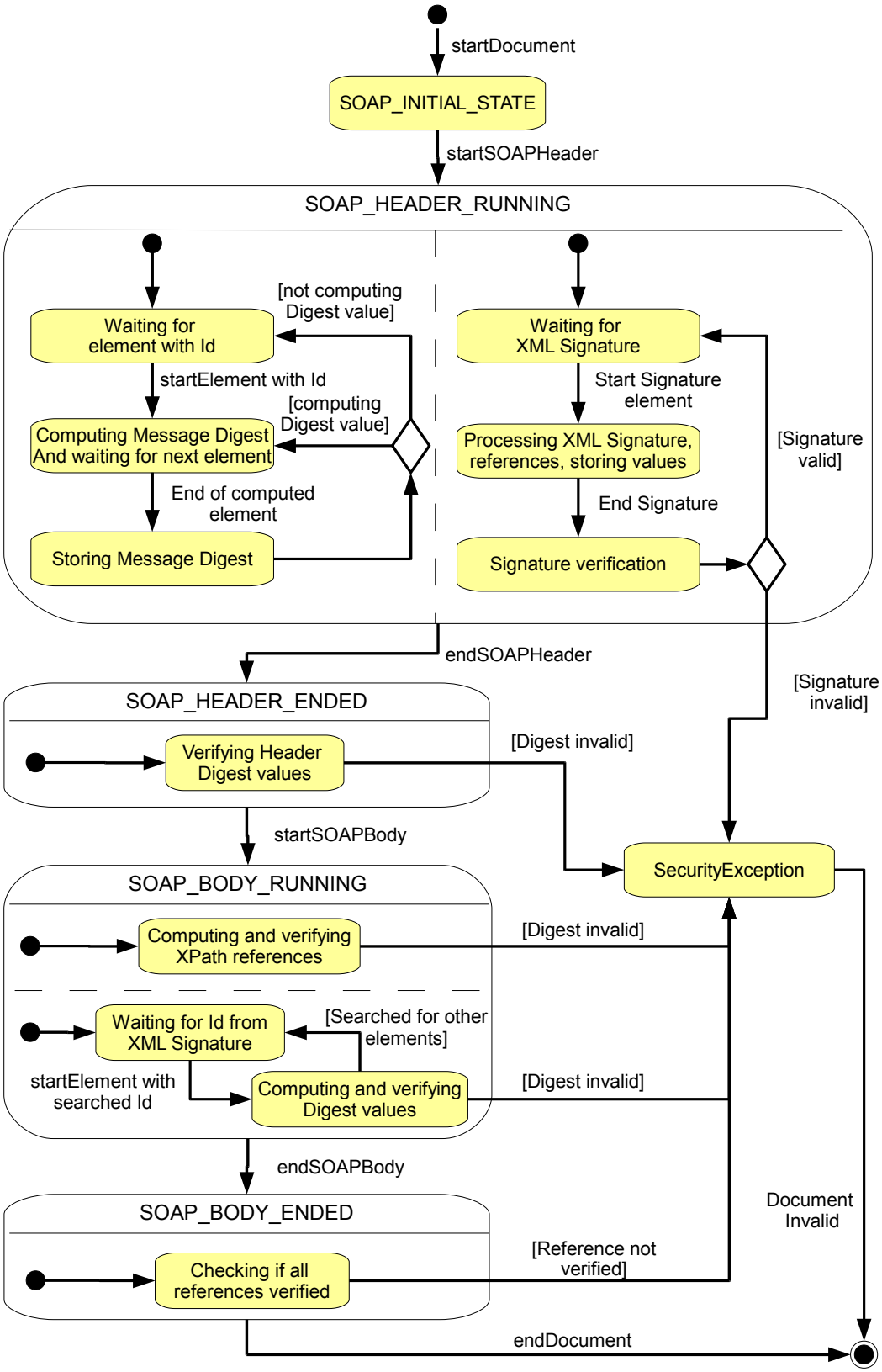

Figure 3.6: XML Signature verification state diagram

left side is depicted handling of references with Ids and on the right side signature processing:

- After initializing of this state, `ReferenceIdHandler` is waiting for elements with Id attributes. If such an element comes, it begins with computing its digest. At the same time it waits for other element with Id attribute. When the end element tag of one of the computed elements appears, the computed digest is saved. Then the reference handling returns to one of the previous states: if no other reference digest is computed, it waits again for another element. If another reference computation is running, the `ReferenceIdHandler` computes its digest till the end.

- `SignatureHandler` is waiting for the `<Signature>` element. With this element is started signature processing: algorithms, references and their properties are stored. At the end the signature value is verified. If it is valid, the `SignatureHandler` waits for another `<Signature>` element. If it is invalid, the `SecurityException` is thrown and the document parsing ends.

- `SOAP_HEADER_ENDED`: When this state comes, it means that all the digests from SOAP header were computed and the properties of XML Signatures were parsed. Therefore the needed references are verified. If one of these references is invalid, the `SecurityException` is thrown.

- `SOAP_BODY_RUNNING`: The proceeding is again divided into two parts. The `ReferenceXPathHandler` handles XPath references (this is more concretely described in the next section). The `ReferenceIdHandler` handles Id references. But in comparison to the `SOAP_HEADER_RUNNING` state, it computes only digests of needed references. The invalid references throw again security exceptions.

- `SOAP_BODY_ENDED`: Before the end of document parsing it is checked, if all the needed references were computed. If yes, the document is valid. Otherwise the `SecurityException` is thrown.

According to the state diagram after throwing a `SecurityException` the SOAP message parsing is interrupted. This is illustrated only for the simplicity. In the WSS Gateway is in fact always a possibility to continue parsing. The decision of parsing interruption is made by the `SecurityExceptionHandler`, which was described earlier in this chapter.

3.5.3 SAX-based XPath Transformation Handling

Along with elements referenced by Ids the WSS Gateway enables processing of XPath transformations. The XPath transformations are based on the XML-Signature XPath Filter 2.0 [RBH02], but they don't cover the full standard. For simplicity only the XPath transformation of the SOAP body is supported and one intersect method is processed (with union and subtract functions XPath filter 2.0 also allows addition and subtraction of XPath results, which is not supported).
Event-based handling of Id references is very simple: the document is being parsed and when an element with the desired Id comes, the digest computing starts. On the other hand,

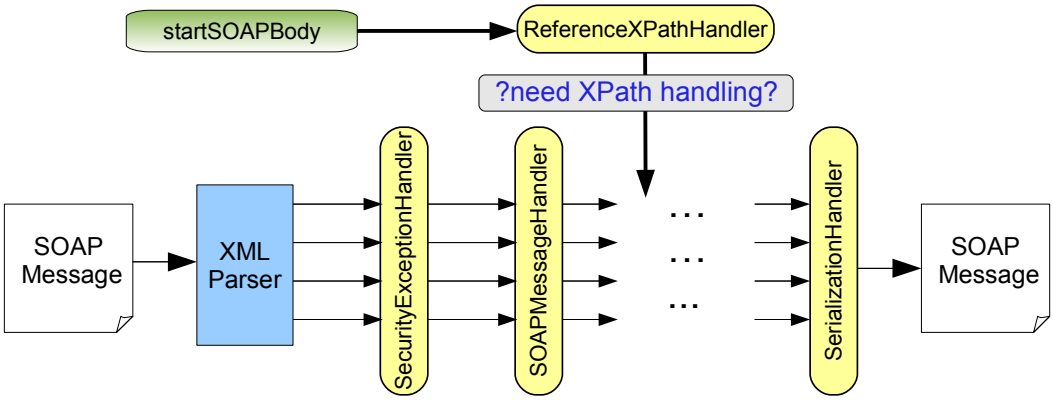

Figure 3.7: The ReferenceXPathHandler is added dynamically to the pipe

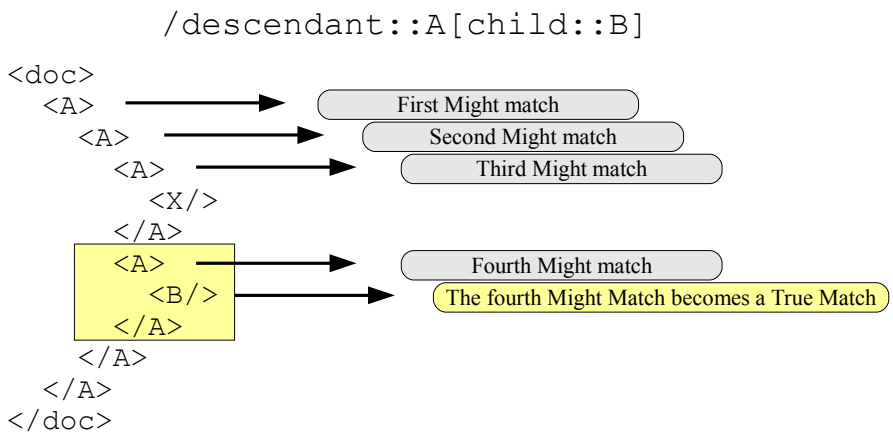

Figure 3.8: True Match and Might Match in XML documents

XPath uses path expressions to select nodes or node-sets in an XML document. The XPath expressions enable to reference each element from any place in the SOAP message. It uses backward and forward navigation axes and predicates to find a specific node or a node that contains a specific value. Backward navigation is impossible using event-based processing, the SAX parser can not return back to previous elements. Predicates cause also problems. For example, during the parsing of an element it is not known whether the element will have a child with the name X.

For the evaluation of XPath expressions in the WSS Gateway is responsible `ReferenceXPathHandler`. The `ReferenceXPathHandler` is added dynamically to the pipe on the beginning of SOAP body parsing only if any XML Signature needs to parse an XPath expressions for its verification (Figure 3.7). The `ReferenceXPathHandler` uses SPEX engine [BCD+06], which is constructed for event-based XPath parsing and offers some useful features. One of these features is rewriting of XPath expressions into forward only navigation axes, which solves the first problem – backward navigation.

SPEX provides also a possibility for processing of predicates containing "future" information. The functionality of this feature can be observed on the expression /descendant::A[child::B]. When this expression is evaluated and an `<A>` element is found, it is not possible to determine, if it is the searched result. This is not known until the `<B>` element is found. Therefore SPEX determines during the parsing two types of matches 3.8:

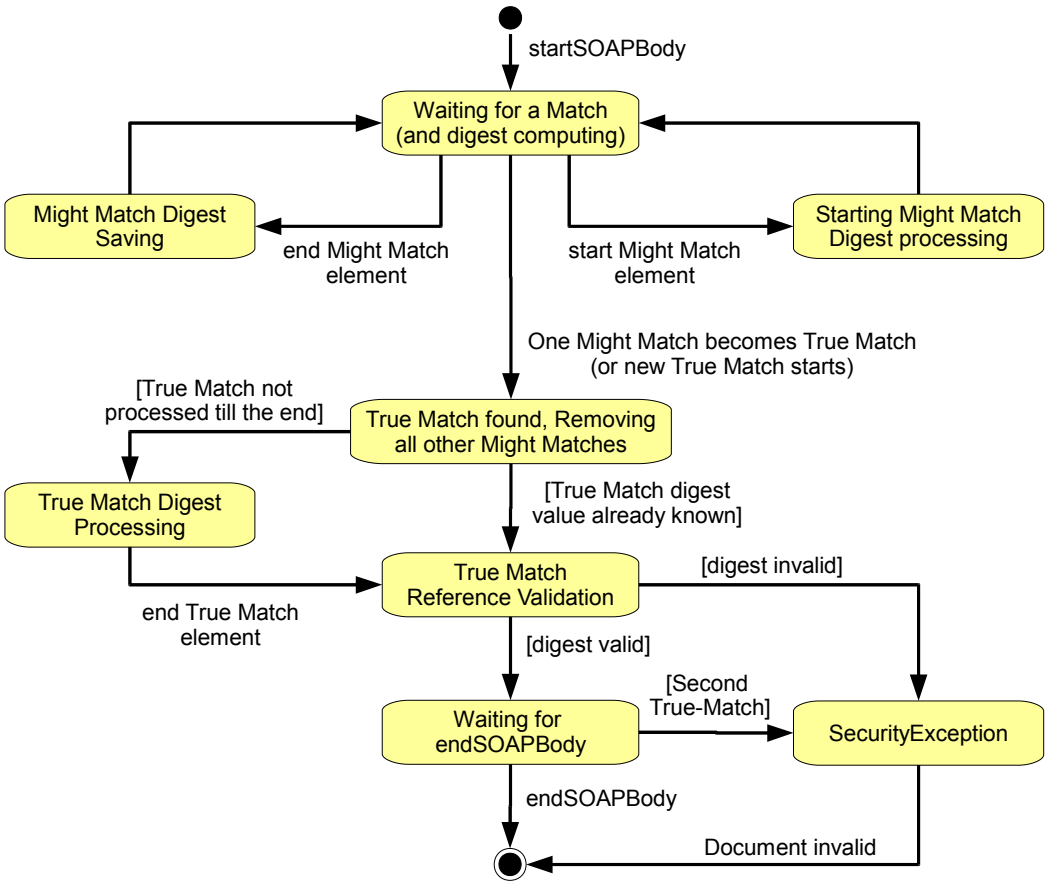

Figure 3.9: State diagram of XPath evaluation

- *Might match* (possible head of result): it can be the result element, but it will be known after processing next document elements.
- *True match* (head of result): the definite result of XPath expression. The True match becomes always one of the assumed Might matches.

As there is no way to go back in event-based XML evaluation, it is first needed to start computing digests of each Might match element. When a True match comes, one appropriate Might match digest is taken and set as True match value. After a True match is found, no other Might match computations are needed.
The state diagram of one XPath expression evaluation is depicted in Figure 3.9. With the start of the SOAP body waits the `ReferenceXPathHandler` for an expression match. If a Might match comes, it is started with digest computation. End of Might match element activates digest saving. These three states can repeat again several times until one of the Might matches becomes a True match. All other Might match values are then deleted and no new digest computation must start. If the digest processing of the found True match has already ended, the digest value is validated. Otherwise the True match element is processed till the end. If the validation of True match element carries a `SecurityException`, the SOAP message is invalid and the processing is interrupted. The `SecurityException` is also thrown, if the second True match before the end of the SOAP message comes. The SOAP message is valid only if one correct True match is found.
This proceeding brings also one disadvantage – an attacker could easily perform DoS at-

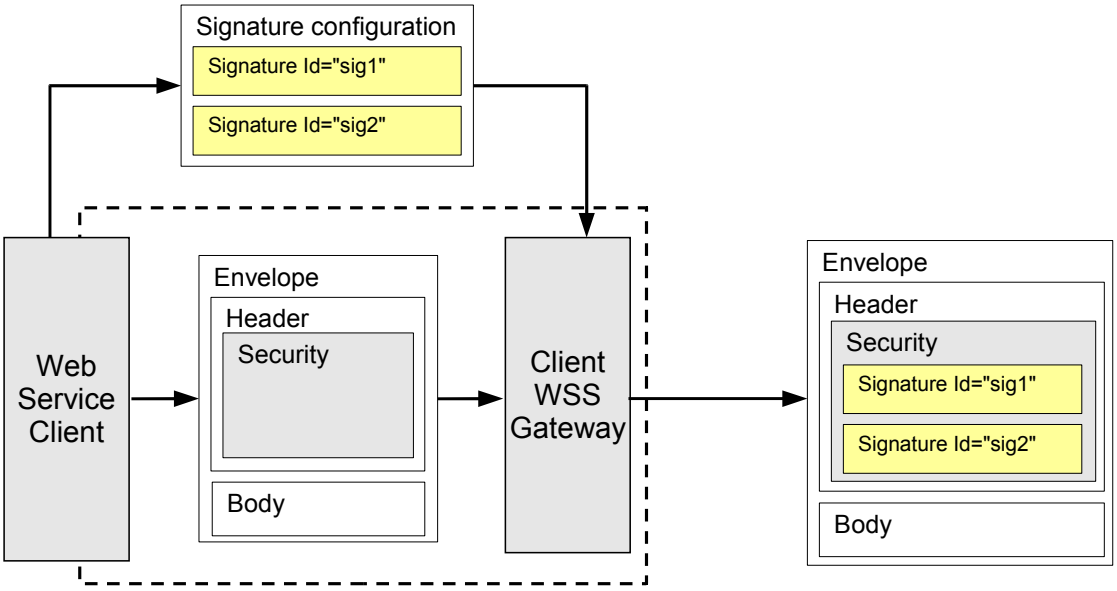

Figure 3.10: Client side WSS Gateway configuration

tacks. Sending a SOAP message containing a reference producing many Might matches (e.g. /descendant::node()[child::X]) forces the WSS Gateway to make many expensive canonicalizations and digest computations. Therefore it is required to set the maximal limit of started Might match digest computations or omit the XPath expressions invoking Might match results.

3.5.4 XML Signature Creation

Server side WSS Gateway handles XML signature verification. XML signature is defined in the SOAP header and contains all data needed for its verification (algorithms, references, digest and signature values). On the other hand Client side WSS Gateway hasn't got any opportunity to get information about signed data from SOAP message. Therefore the Client side WSS Gateway has to be configured from an external configuration file.
In WS-Security for this purpose is used WS-Security Policy [KL07]. This thesis does not cover the WS-Security Policy standard and only a simple XML configuration file is used (shown in the next Chapter in Listing 4.1). The configuration file is passed to the WSS Gateway before its processing starts (Figure 3.10). It includes data about signatures, their references and algorithms. When a configured WSS Gateway gets a SOAP message, it computes references and inserts XML signatures into the SOAP header according the configuration. As the elements are referenced by Id or XPath the client must be careful that the SOAP message includes elements specified in the configuration file. Otherwise the XML signature could not be computed.

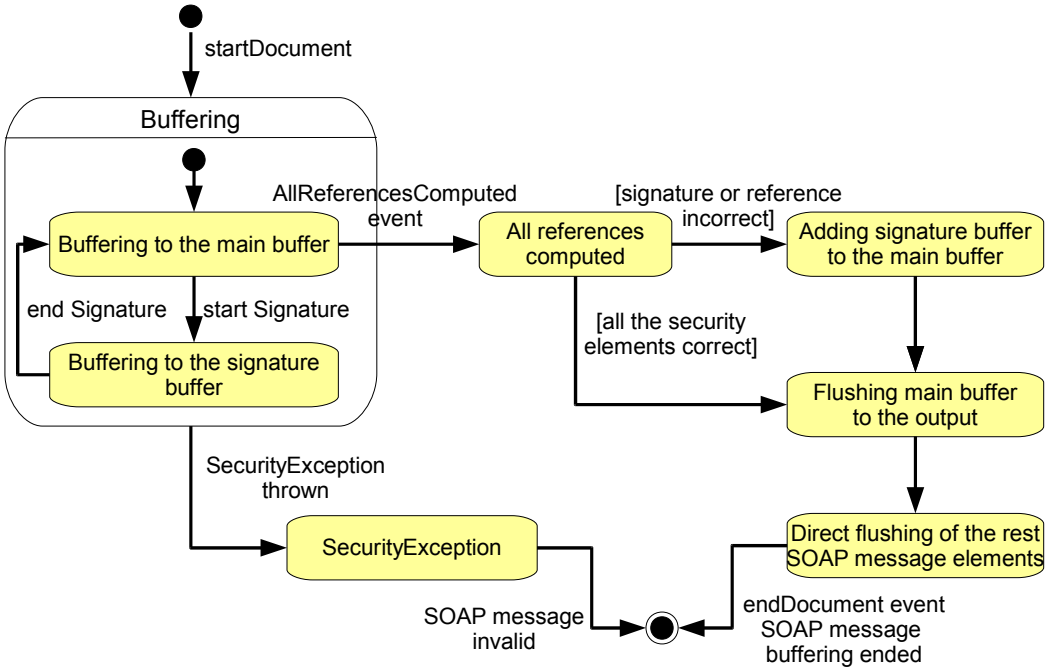

Figure 3.11: State diagram of SOAP message serialization

3.6 SOAP Message Serialization

The common usage of WSS Gateway is for verifying (or creating) and forwarding signed SOAP messages. The SOAP messages are accordingly parsed into events, processed and put back together. To restore back the parsed XML data, a serialization module is needed. The Serialization module stands at the end of pipeline. It collects basic SAX events and constructs again element tags. These are forwarded to the file or another application. Each WSS Gateway side has got its own serialization module so there exist client and server `SerializationHandlers`.

The state diagram illustrated in Figure 3.11 shows the functionality of serialization module on the server side by SOAP message verifying. The server side serialization module has two buffers – main and signature buffer. On the beginning each SOAP message event is collected to the main buffer. When a signature element starts, the incoming elements are saved to the signature buffer. Each XML Signature is stored separately. The end of last signature element switches buffering again to the main buffer so in the signature buffers are kept only signature relevant elements. Saving to the main buffer lasts until the last reference element comes. Verification of the last element invokes `allReferencesComputed` event, which brings the proceeding to another state. If all the references are correct, the SOAP message is validated and the main buffer can be flushed to the output (sent to the Web Service Server). As the document is valid, the remaining elements must not be held in the memory, they are flushed directly during parsing.

A special scenario can occur if the `SecurityExceptionHandler` allows some SecurityExceptions regarding XML Signatures (e.g. it allows to have one incorrect reference or signature value). In this case when the `allReferencesComputed` event comes the invalid XML Signatures are taken from the signature buffers and put to the main buffer so they can be sent to a Web Service Server. The Web Service Server can then make a decision what to do next with

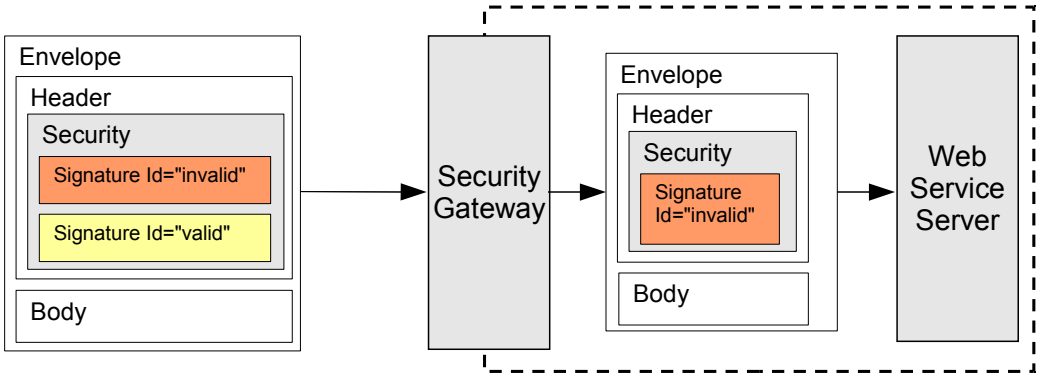

Figure 3.12: SOAP message with incorrect XML Signature

these incorrect SOAP messages. An example of such SOAP message handling is depicted in Figure 3.12.

4 Implementation

This chapter contains information about implementation of the Event-based WSS Gateway. First it is discussed the structure of packages in the application. Then it is shown the handling of Exclusive XML Canonicalization with an event-based approach. In the last section there are described the concrete implementation of WSS Gateway and each module in the pipe.
The WSS Gateway was developed using Java SE Development Kit 6. The JDK 6 addresses many standard security facets such as key generation and management, signing and encrypting a content or certificate management [Mic06a]. Therefore there was no need to add an external security provider (e.g. Bouncy castle [con09]) for this test implementation. The implementation includes only the SPEX engine mentioned in the previous chapter and the Apache Commons Codec package [Apa09], which is used for Base64 encoding.

4.1 Package Hierarchy

The WSS Gateway classes are organized in several packages. Their structure is depicted in the diagram in Figure 4.1 (for simplicity it does not include all the used classes and package relations). The base of the structure forms de.rub.nds.xml.wessega, which includes all the WSS Gateway packages:

- `main`: Contains the `WSSecurityGateway` class. With this class the whole WSS Gateway application can be started. According to the parameter passed to the class constructor can be decided, if the WSS Gateway operates on the *server* side or on the *client* side.
- `utils`: Includes standard utilities used by all the classes in the application.
- `pipeline`: There can be found classes representing all the modules in the pipe.
- `signature`: Includes classes responsible for signature handling: building and parsing XML Signatures and storing their data.
- `xpath`: Holds classes controlling the connection between WSS Gateway and SPEX engine and evaluating the data.
- `exceptions`: Contains security exceptions that could be thrown by the application.

The classes included in the packages signature and utilities provide the whole application with useful features. Therefore the concrete explanation of their structure and functionality

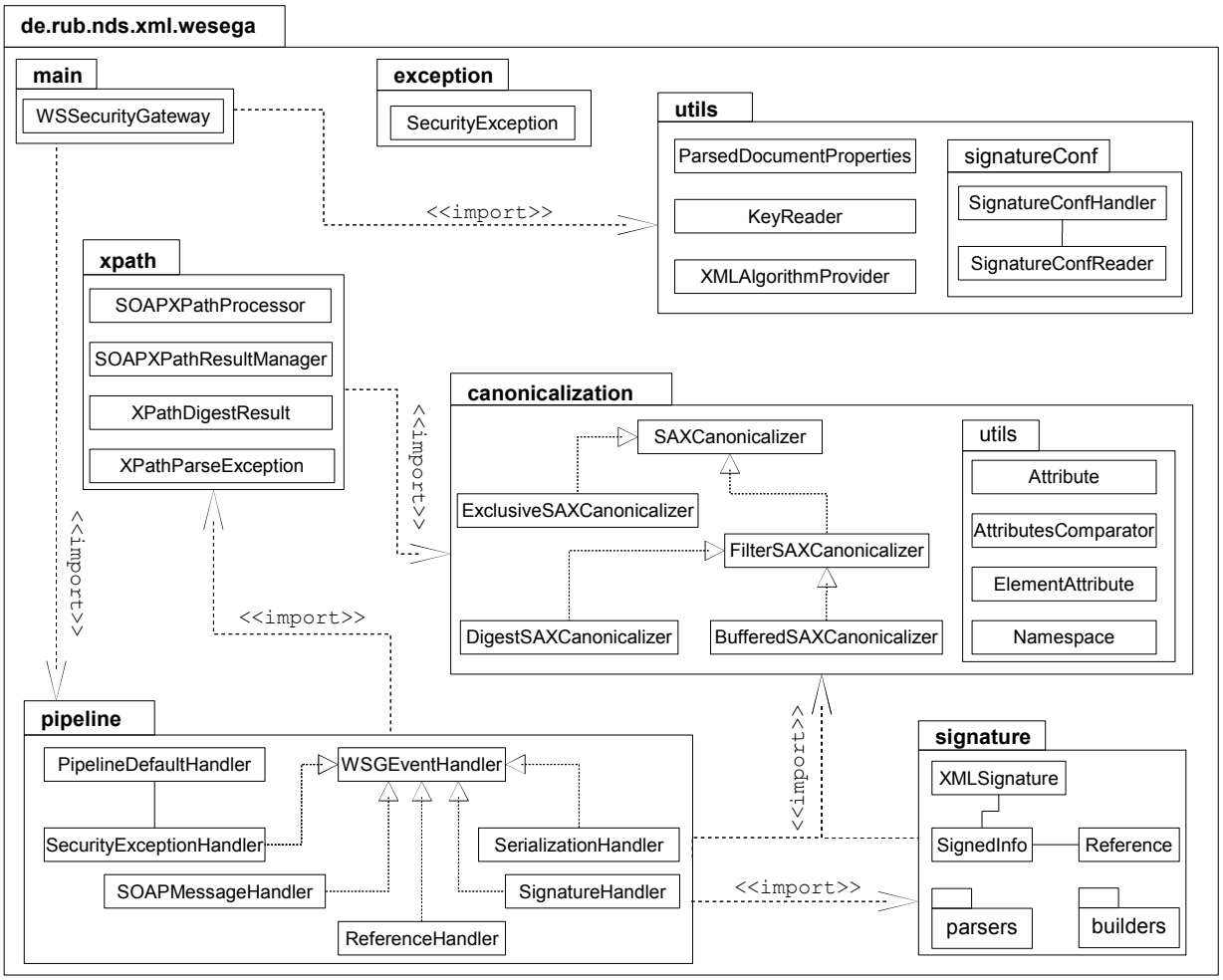

Figure 4.1: The WSS Gateway Package Diagram

follows in this section, before the concrete application modules will be explained.

4.1.1 Utilities

The package `de.rub.nds.xml.wessega.utils` contains classes `ParsedDocumentProperties`, `KeyReader` and `XMLAlgorithmProvider`. It includes also the package `signatureConf`, which holds two additional classes `SignatureConfHandler` and `SignatureConfReader` and therewith provides the client side WSS Gateway with a feature for reading XML configuration data.

ParsedDocumentProperties

`ParsedDocumentProperties` class is a central storage of data collected by different modules. New object of this type is generated at the begin of each document parsing. All the modules

in the pipe get a reference to this new generated object. This way the modules can exchange their information.
This proceeding is against standards of the Event Pipeline Pattern, where every information has to flow through the pipe. But having a centralized object makes the implementation clear and avoids data duplicity in different modules.
In `ParsedDocumentProperties` are stored for example XML Signatures, their references with algorithms or actual position in the document (SOAP header, body...).

KeyReader

During the application running, it is assumed that the public keys have already been exchanged and the WSS Gateway is in the possession of a correct key pair. The key pair is stored in a `*.keystore` file. The class `KeyReader` is implemented to read this file and get key data. It provides the application with functions `readKeyFile` (for reading .keystore file), `getPrivateKey` and `getPublicKey` (for getting the keys after reading the file). As the application needs only one `KeyReader`, the class is implemented as a *Singleton* [FF04].

XMLAlgorithmProvider

This class creates a mapping between java security algorithm names and XML Security algorithm URI identifiers according to the XML Security Algorithm Cross-Reference [HRY09]. In the class are defined only the algorithms used in the WSS Gateway. The algorithms are stored in static arrays as shown below:

```
/** Signature algorithms and their URI */
private static final String[][] signatureAlgorithms = {
  {"http://www.w3.org/2000/09/xmldsig#rsa-sha1", "SHA1withRSA"},
  {"http://www.w3.org/2001/04/xmldsig-more#rsa-md5", "MD5withRSA"},
  {"http://www.w3.org/2000/09/xmldsig#dsa-sha1", "SHA1withDSA"}
};
```

The public static functions `getSignatureAlgorithm` and `getSignatureAlgorithmURI` allow then access the security algorithm name pairs. If the requested algorithm name or URI isn't included in the array, the `XMLAlgorithmProvider` throws a security exception.

SignatureConfReader

Client side WSS Gateway needs to get information about signed XML data from an external configuration file. The configuration file contains information about XML Signature properties and all the elements that have to be signed. It is loaded on the WSS Gateway immediately after its start. The task of the `SignatureConfReader` is to extract the XML Signature configuration. This configuration is then applied on each incoming SOAP message. Therefore it has to be assured that incoming SOAP messages contain data needed for XML Signature creation (Id or XPath references point to existing elements).
The configuration file is an XML document and its structure is very simple. An example is

```
<signature-conf>
   <signature id="Signature1">
      <SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#rsa
         -sha1"/>
      <Reference URI="#Ref1">
         <DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#
            sha1"/>
      </Reference>
   </signature>
   <signature id="Signature2">
      <SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#dsa
         -sha1"/>
      <Reference xpath="/child::VehiclePartLookup/child::Shared">
         <DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#
            sha1"/>
      </Reference>
   </signature>
</signature-conf>
```

Listing 4.1: Example of a signature configuration file

shown in Listing 4.1. It contains one root element named `<signature-conf>`. Its children are `<signature>` elements. In the example the first signature uses RSA with SHA1. It signs the element with Id "Ref1". For computing message digest it is used SHA1. The second signature uses DSA with SHA1 algorithm. The element referencing is done through XPath. As the reference handler begins searching in body, full path to this element in SOAP message would be: `/Envelope/Body/VehiclePartLookup/Shared`. For creating canonical form the default Exclusive XML Canonicalization without comments is applied.

4.1.2 Signature Handling Utilities

Signature handling utilities are organized in the package `de.rub.nds.xml.wessega.signature`. This package contains three classes for storing XML Signature data: `XMLSignature`, `SignedInfo` and `Reference`. The Reference class contains data describing the Reference element: Id, digest method, canonicalization method and XPath transformation expression. The `SignedInfo` class includes besides the list of `Reference` objects the information about the canonicalization method and the signature method. The `SignedInfo` is included in the `XMLSignature` class, which contains also the XML Signature Id and the signature value. The best description of these classes and their attributes gives the class diagram depicted in Figure 4.2. Additionally to the private attributes the classes own the appropriate getters and setters methods.
Each of these classes is used in the equally-named parser and builder class.

Parsers

Three classes declared in the package `signature.builders` are dedicated for the server side parsing of XML Signatures. Every class implements the `BasicXMLElement` interface so it can

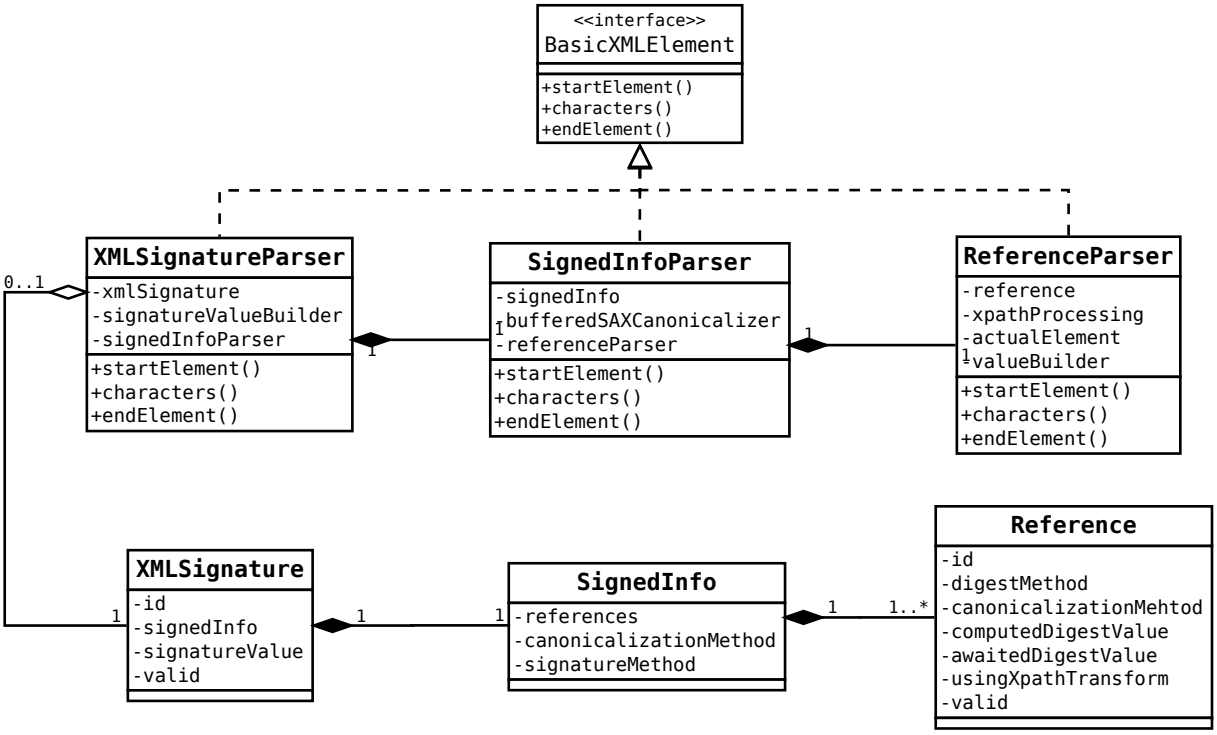

Figure 4.2: The class diagram of the signature handling classes

get and process the SAX events `startElement`, `characters` and `endElement`. The parsing is controlled by the `XMLSignatureParser`, which begins working with the start of `<Signature>` element. It contains one `XMLSignature` attribute where the collected data are stored. When a `SignedInfo` element comes the parsing is managed by the `SignedInfoParser`. This class collects information about canonicalization and signature methods and stores them into its `SignedInfo` attribute. Along with collecting data the canonical form of the `SignedInfo` block is being buffered. References are then handled by a `ReferenceParser`. The output of the `ReferenceParser` is a list of `References`. It is added to the `SignedInfo` object, which is then put into the `XMLSignature`. At the end of XML Signature parsing the `XMLSignature` object is put to validation process.

Builders

The package `signature.builders` contains three classes for building of XML Signatures in the client side WSS Gateway. Each class has only one public static function called build. To this function is passed an appropriate class attribute from the package signature. The task of the build function is to compute signature and digest values and construct a correct XML structure from the incoming parameter. Outputs of the build functions are in the canonical form so there is no canonicalization after building needed. An example of the `XMLSignatureBuilder` can be seen in Listing 4.2.

```
public class XMLSignatureBuilder {

  public static StringBuilder build(XMLSignature signature) {
    StringBuilder sb = new StringBuilder("\n<ds:Signature Id=\"").
        append(signature.getId()).append("\" xmlns:ds=\"http://www.w3.
        org/2000/09/xmldsig#\">");

    SignedInfoBuilder signedInfoBuilder = new SignedInfoBuilder(
        signature.getSignedInfo());
    sb.append(signedInfoBuilder.build());

    sb.append("\n<ds:SignatureValue>\n");
    sb.append(signedInfoBuilder.getSignatureValue()).append("\n");
    sb.append("</ds:SignatureValue>\n");
    sb.append("</ds:Signature>");

    return sb;
  }

}
```

Listing 4.2: XMLSignatureBuilder.java

4.2 Exclusive XML Canonicalization

Exclusive XML Canonicalization is used in more pipeline modules in several ways. In modules handling references it is needed for direct digest computing. In signature handler it is required to put `<SignedInfo>` element into the canonical form and the buffered result must be passed to the signature validation method. Also it is very useful to have a possibility to buffer and compute message digest in the same time (e.g. by testing) or add another canonicalization method. Because of these requirements the *Decorator Pattern* [FF04] [EGV95] was used.
The Decorator Pattern (Figure 4.3) attaches additional functionality to an object dynamically, by wrapping a component with decorators. At the core of the implementation lies a concrete component, `ExclusiveSAXCanonicalizer`. It accepts basic SAX events, processes them and outputs the events as strings in canonical form. The canonical form of each event serves as an input for decorators – `BufferedSAXCanonicalizer` and `DigestSAXCanonicalizer`. `BufferedSAXCanonicalizer` puts the canonical form of an event into its buffer. `DigestSAXCanonicalizer` computes new digest value.

4.2.1 Utilities

Before describing the functionality of concrete canonicalization classes the canonicalization utilities are introduced. They are stored in the package `de.rub.nds.xml.wessega.canonicalization.utils`. There are included two classes representing element attributes: `Namespace` and `Attribute`. Class `Namespace` relates to an element declared namespace and class `Attribute` to an element declared attribute. Both of them implement the interface `ElementAttribute` as depicted in Figure 4.4.
During the canonicalization process these element attributes have to be sorted. There-

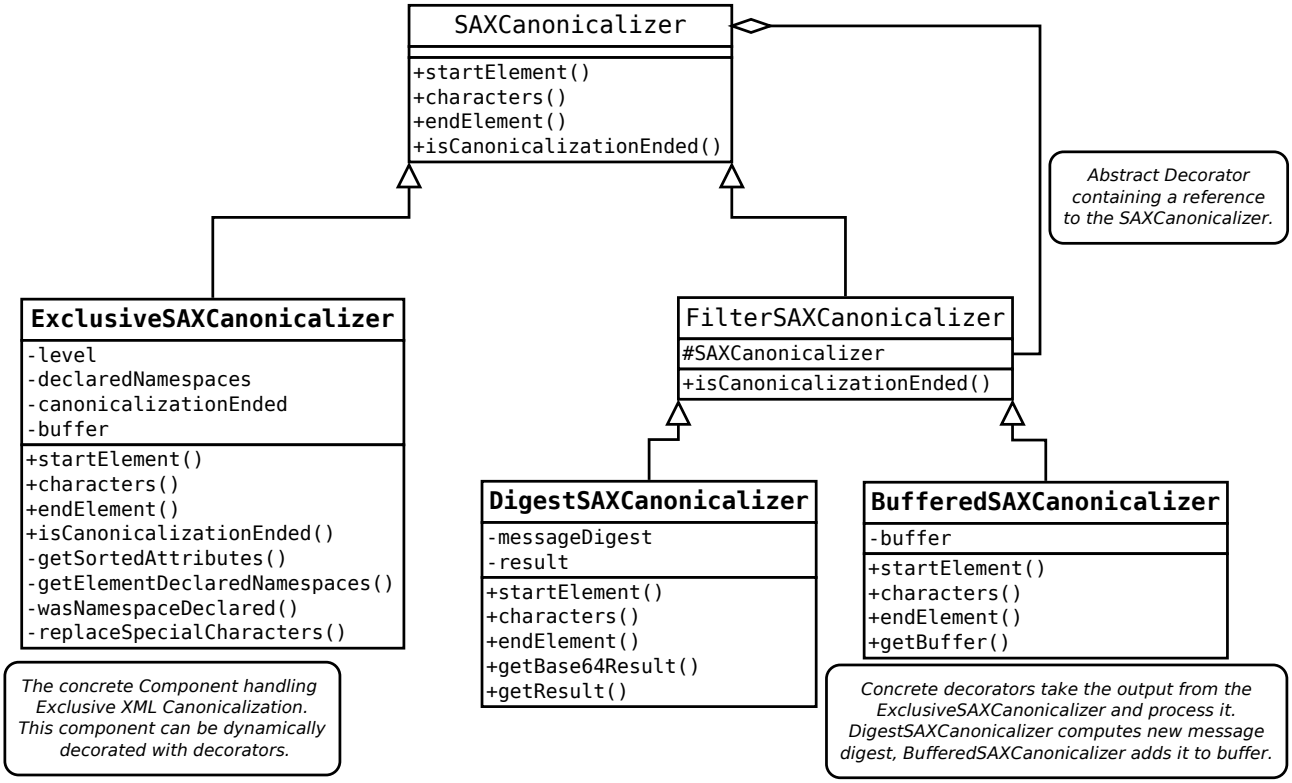

Figure 4.3: Classes handling Exclusive XML Canonicalization

fore the java function `sort` from the class `java.util.Collections` is used. This function takes two parameters: the list of `ElementAttributes` and an `AttributesComparator`. The `AttributesComparator` is a class implementing the `Comparator` interface and including the function compare, which compares two element attributes for order. It returns `-1`, `0` or `1` as the first argument is less than, equal to, or greater than the second. The implementation of this class is shown in Listing 4.3.

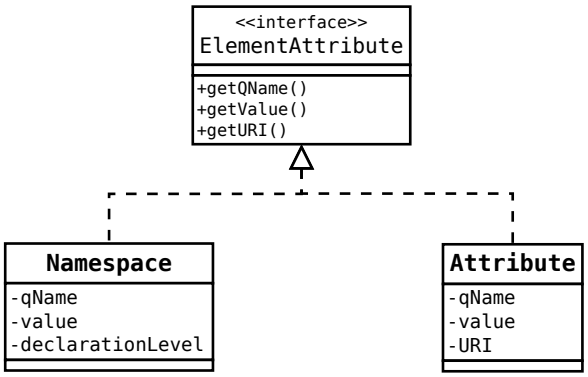

Figure 4.4: Canonicalization utilities classes

```
public class AttributesComparator implements Comparator<
    ElementAttribute> {
  public int compare(ElementAttribute a1, ElementAttribute a2) {
    // if the first attribute declares xmlns, place it before second
        attribute
    if( (a1.getQName().startsWith("xmlns")) && !(a2.getQName().
        startsWith("xmlns")) ) {
      return -1;
    }
    // if the second attribute declares xmlns, place it before first
        attribute
    if( !(a1.getQName().startsWith("xmlns")) && (a2.getQName().
        startsWith("xmlns")) ) {
      return 1;
    }
    // if both attributes declare namespace, sort it according qNames
    if( (a1.getQName().startsWith("xmlns")) && (a2.getQName().
        startsWith("xmlns")) ) {
      return a1.getQName().compareTo(a2.getQName());
    }
    // if none of two attributes declares xmlns, compare URIs
    if(a1.getURI().compareTo(a2.getURI()) != 0) {
      return a1.getURI().compareTo(a2.getURI());
    }
    // if none of two attributes declares xmlns and their URIs are
        equal, compare their qNames
    return a1.getQName().compareTo(a2.getQName());
  }
}
```

Listing 4.3: AttributesComparator.java

4.2.2 ExclusiveSAXCanonicalizer

The `ExclusiveSAXCanonicalizer` implements three SAX events: `startElement`, `characters` and `endElement`. It has got four basic attributes:

- `level`: The depth in canonicalized XML tree. Each start-tag increases it by one, each end-tag decreases by one.
- `declaredNamespaces`: Namespaces already declared during the canonicalization process. Each namespace has got its declaration level.
- `canonicalizationEnded`: Boolean variable telling, if the canonicalization process has been already ended.
- `buffer`: Buffer for constructing canonical form of each event.

Because of the namespaces declaration complexity and attributes sorting, the most expensive part of canonicalization is start-tag processing. When the `ExclusiveSAXCanonicalizer` gets a `startElement` event, it handles as follows:

1. It constructs a list of namespaces visibly utilized by the current element. From these namespaces are then chosen the namespaces, that haven't been already declared (or

that have been redeclared). For this purpose are dedicated `getElementDeclaredNamespaces` and `wasNamespaceDeclared` methods.

2. The list of all element attributes (inclusive newly declared namespaces found in step 1) is created and special characters in the attributes are replaced (with the method `replaceSpecialCharacters`).

3. The attributes are sorted with the method `getSortedAttributes`, which uses the canonicalization utilities.

4. The canonicalized element is built: `"<" + qName + sorted attributes + ">"`.

5. Level (XML tree depth) is increased.

As opposed to `startElement`, the handling of `characters` and `endElement` events is very simple. In `characters` event are only replaced special characters. In `endElement` event is constructed an appropriate end tag: `"</" + qName + ">"`.

4.2.3 Use of Exclusive Canonicalization

The decorators `BufferedSAXCanonicalizer` and `DigestSAXCanonicalizer` implement SAX events `startElement`, `characters` and `endElement` and use the functionality of `ExclusiveSAXCanonicalizer`. When they get one of these events, they send it to the `ExclusiveSAXCanonicalizer`, that computes the event's canonical form. The canonical form is then put to the buffer or used to compute new message digest. The following code demonstrates the usage of the Exclusive SAX Canonicalization:

1. Initialization: The canonicalization is initialized. First is created the `ExclusiveSAXCanonicalizer` object. This object is then wrapped by `BufferedSAXCanonicalizer` and `DigestSAXCanonicalizer`.

```
// ...
ExclusiveSAXCanonicalizer exclusiveSC =
      new ExclusiveSAXCanonicalizer();
BufferedSAXCanonicalizer bufferedSC =
      new BufferedSAXCanonicalizer(exclusiveSC);
DigestSAXCanonicalizer digestSC =
      new DigestSAXCanonicalizer(bufferedSC);
// ...
```

2. `DigestSAXCanonicalizer` as the wrapping object sends SAX events to the `BufferedSAXCanonicalizer` and `ExclusiveSAXCanonicalizer`. This simple command computes event's canonical form, adds it to the buffer and updates message digest.

```
// ...
digestSC.startElement(uri, localName, qName, attributes);
// ...
```

3. The results are then collected this way:

```
// ...
```

```
StringBuilder sb = bufferedSC.getBuffer();
byte[] digest = digestSC.getResult();
// ...
```

Benefit of the Decorator pattern is that it offers a possibility of extensions. In the future there could be easily added new components or decorators. For example the implementation could also handle Exclusive Canonicalization with Comments. This would be done only by adding a new `ExclusiveSAXCanonicalizationWithComments` component, which implements all the abstract components' methods. The Decorator pattern is in widespread use. Probably the most famous usage is in Java I/O classes [Mic06b].

4.3 Event Pipeline Pattern Application on WSS Gateway

On the implementation of Event-based WSS Gateway was applied the *Event Pipeline Pattern*. The UML diagram can be seen in Figure 4.5. At the core stands the abstract class `WSGEventHandler`. The `WSGEventHandler` owns basic implementation of all events used in the pipeline. The basic implementation of an event is very simple: it is checked, if there exists any next module and if yes, the event is passed to it. This way it is ensured the default flow of events through the pipe.

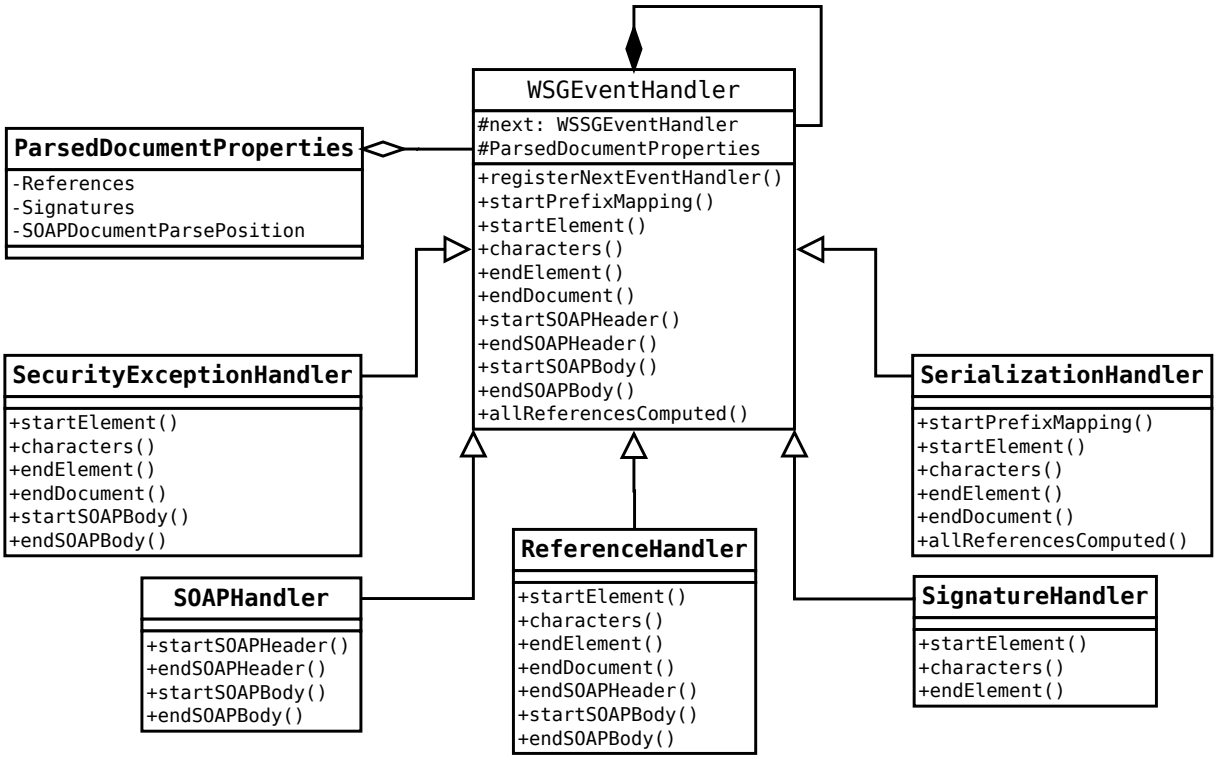

Figure 4.5: Event Pipeline Pattern application on WSS Gateway

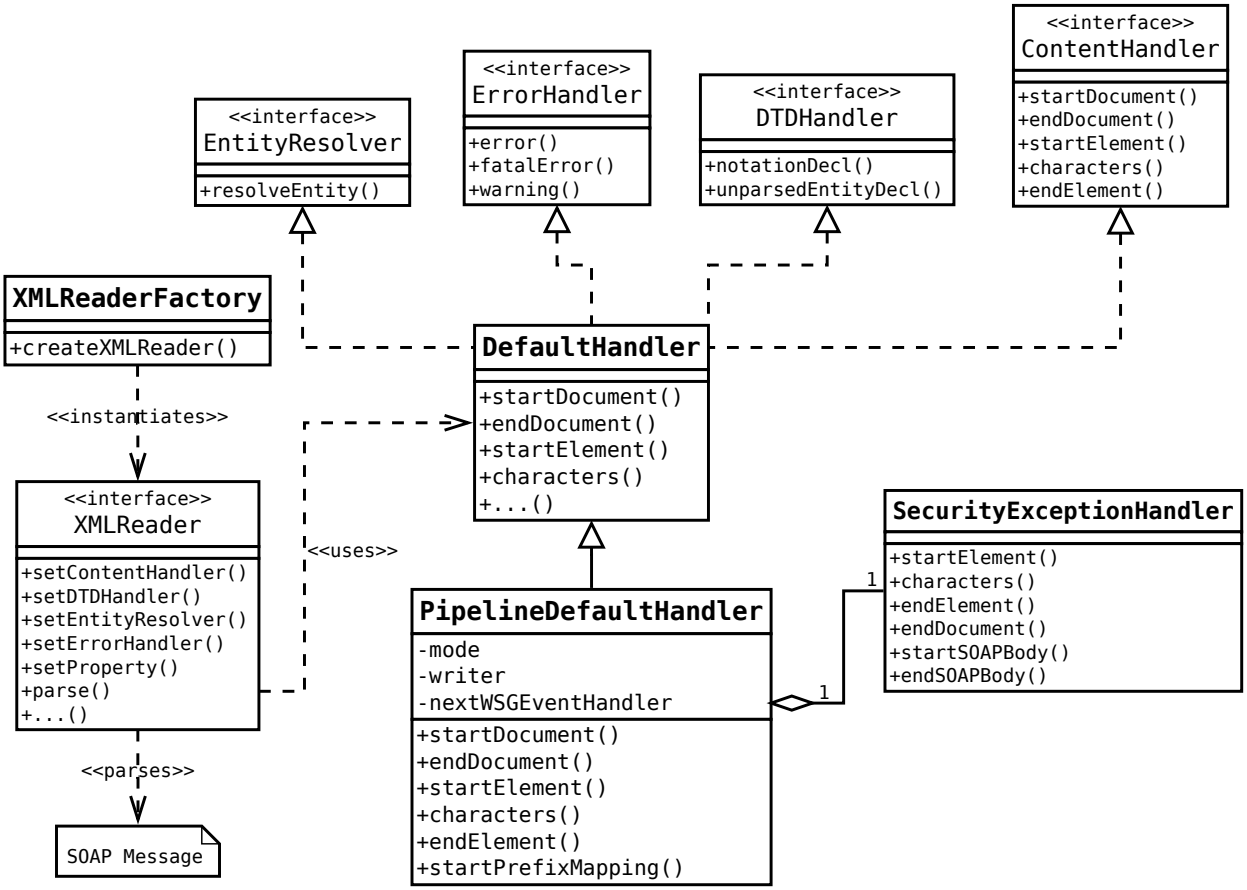

Figure 4.6: SAX Parser connection to the pipeline

The `WSGEventHandler` also contains two class members. Member `next` points to the next module in the pipe. `ParsedDocumentProperties` contains all data about the parsed XML document.
Every pipeline module extends the `WSGEventHandler`. This enables the modules to get information about generated events. If the module needs to process an event, it overrides the appropriate method and adds the event-handling new behavior. At the end of event-handling the event is passed to the next handler.

4.3.1 Connection to the SAX Parser

The structure of SAX parser and its connection with WSS Gateway is depicted in Figure 4.6. There are shown these SAX classes and interfaces:

- `XMLReaderFactory`: The factory instantiating a class implementing an `XMLReader` interface.
- `XMLReader`: Interface offering parse methods for parsing input data. The handling of input data is done by the `DefaultHandler` class.
- `ContentHandler`: Interface for document parsing events such as `startDocument`, `end-`

`Document`, `startElement`.

- `ErrorHandler`: Interface for handling errors and warnings.
- `DTDHandler` and `EntityResolver`: Interfaces for handling DTD and resolving entities.
- `DefaultHandler`: Class implementing four handler interfaces. All its events return default null values. To add a concrete behavior for XML document parsing this class has to be extended.

The mentioned classes and interfaces are contained in the SAX packages. To include new parsing behavior the class `PipelineDefaultHandler` has been created. This class extends the `DefaultHandler` and therewith is responsible for the parsing of XML documents. If it gets a `startDocument` event, it creates a new pipe according to the mode parameter. The mode indicates the WSS Gateway side. It is set in the `PipelineDefaultHandler` constructor to the value `"server"` or `"client"`. This differentiation is needed because the client and server side WSS Gateways do not use the same pipeline modules.
The `PipelineDefaultHandler` is then connected to the pipe through the first pipe module – `SecurityExceptionHandler`. Through this module it sends to the pipe all the handled events: `startElement`, `characters`, `endElement` and `startPrefixMapping`. If the `endDocument` event comes, the whole pipe is deleted.

4.3.2 SecurityExceptionHandler

The `SecurityExceptionHandler` is the first module in the pipe. It can process all the possible security exceptions thrown by other modules:

- `IllegalSOAPStateException`: The SOAP message doesn't conform the recommendation.
- `NoSuchAlgorithmException`: The `XMLAlgorithmProvider` is responsible for converting XML Security algorithm's URI to the algorithm's java name. If the searched algorithm is not found, this exception is thrown.
- `ReferenceMoreResultsException`: Thrown when a reference in the XML Signature points to more elements.
- `ReferenceNoResultException`: Thrown when a reference in the XML Signature doesn't reference any element.
- `ReferenceValidationException`: Reference digest invalid.
- `SignatureValidationException`: Signature value invalid.
- `UnsupportedOperationException`: Thrown when the desired functionality (defined by the standard) is not supported. For example the functionality of the XPath filter's union or subtract methods is not implemented.
- `XPathParseException`: The XPath expression is incorrect and could not be parsed.

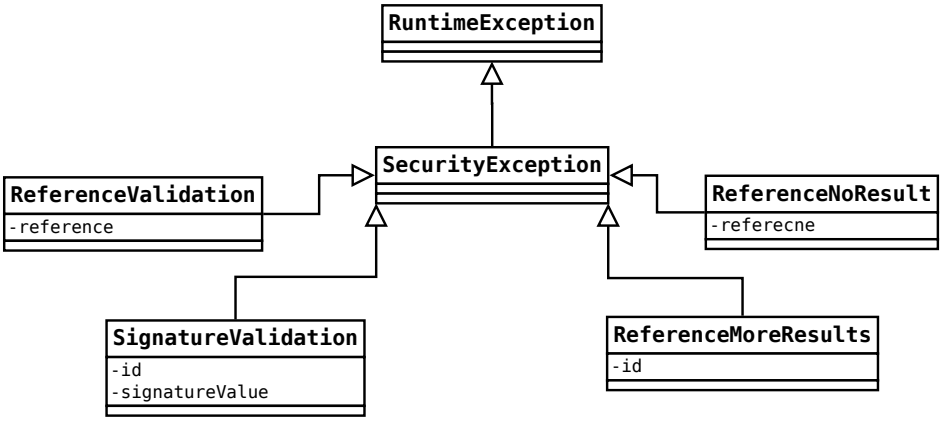

Figure 4.7: Security Exceptions

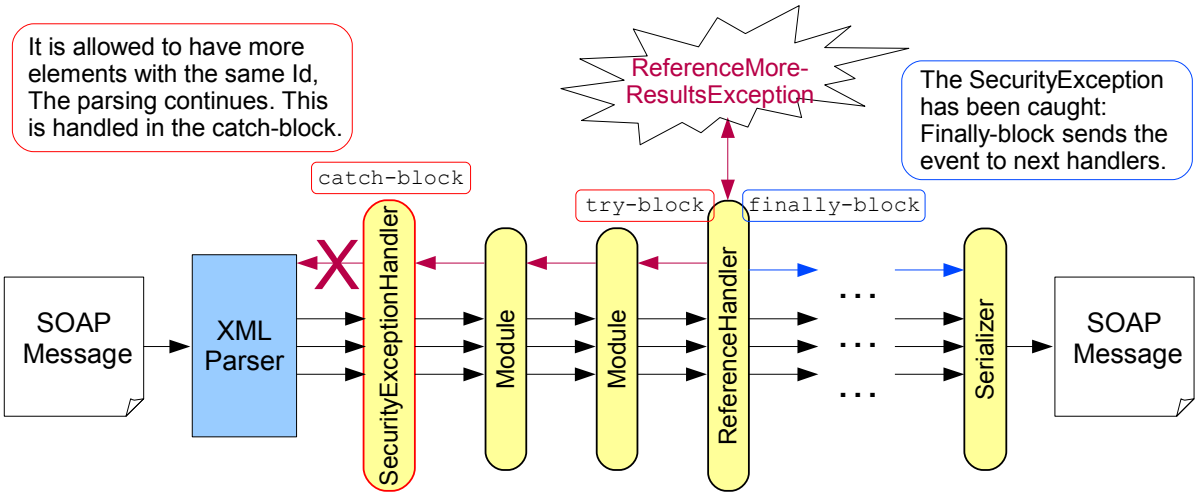

Figure 4.8: SecurityExceptionHandler handling the ReferenceMoreResultsException

UML diagram containing the security exceptions described above is shown in Figure 4.7. As these classes inherit from java `RuntimeException`, the exceptions are all unchecked.
An example of using the `SecurityExceptionHandler` gives handling `ReferenceMoreResults-Exception`. This exception indicates that more elements are using the same Id (this is usual by wrapping attacks). Even thought, the user can decide to ignore this exception and only log a warning (Figure 4.8). Therefore, it must be implemented the event throwing this exception in `SecurityExceptionHandler` class and the behavior after catching it. This could be done by the following code:

```
public void endElement(String uri, String localName, String qName) {
  try {
    if(next != null) {
      next.endElement(uri, localName, qName);
    }
  } catch(ReferenceMoreResultsException rmre) {
    System.out.println("There are two elements with the same Id, but we
          continue with parsing.");
  }
}
```

Throwing a security exception interrupts the event processing in the middle of the pipe. When the exception is caught in the `SecurityExceptionHandler`, it is important, that this event is processed also by other handlers till the end (e.g. the serialization handler on the end must get all the events, otherwise the generated document is incomplete). For this reason are exception-generating functions in handlers put in `try-finally` blocks. This ensures not only sending exceptions back to the `SecurityExceptionHandler` (where the `catch-block` could be applied) but also passing the event always to the next handler (in `finally` block):

```
public void endElement(String uri, String localName, String qName) {
  try {
    // ...
    can.endElement(uri, localName, qName);
    if(can.isCanonicalizationEnded()) {
      // this function can throw an exception
      parsedDocumentProperties.putComputedReference(ref.getKey(), can.
          getResult());
  } finally {
    // this ensures the next handler processes this event
    super.endElement(uri, localName, qName);
  }
}
```

4.3.3 SOAPMessageHandler

The `SOAPMessageHandler` checks the current location in SOAP message during XML parsing. There are five possible states: `SOAP_INITIAL_STATE`, `SOAP_HEADER_RUNNING`, `SOAP_HEADER_ENDED`, `SOAP_BODY_RUNNING`, `SOAP_BODY_ENDED`. The states must follow exactly in this series. Otherwise an `IllegalSOAPStateException` is thrown. This ensures the SOAP message has got the correct form and prohibits attacks where e.g. two SOAP bodies are used.
When the correct state change comes, it is written to the `ParsedDocumentProperties` object. At the same time a new event is generated. `SOAPMesageHandler` generates these message events: `startSOAPHeader`, `endSOAPHeader`, `startSOAPBody`, `endSOAPBody`. Therefore if some handlers want to use these events, they have to be put in the pipe behind this handler.

4.3.4 SignatureHandler

The `SignatureHandler` is inserted only to the pipe of the server side WSS Gateway application. It is responsible for processing of `<Signature>` element in the header of SOAP messages. During parsing of this element it buffers the canonical form of `<SignedInfo>` created by the `ExclusiveCanonicalizer`. It collects also data (references, digest methods, ...) and stores them in `ParsedDocumentProperties`. This is all done with the help of `XMLSignatureParser`. At the end of the `<Signature>` element, it verifies the signature value. If the signature value is incorrect, a `SignatureVerificationException` is thrown.
There are two types of references the `SignatureHandler` can process and send to `ParsedDocumentPropere`
Id references and *XPath* references. The Id reference points to an element in the document

```
<ds:Reference URI="#Ref1">
   <ds:Transforms>
      <ds:Transform Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"
         />
   </ds:Transforms>
   <ds:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/
      >
   <ds:DigestValue>YyUHjixfVsF5dLWALRY7Dk1gmtg=</ds:DigestValue>
</ds:Reference>
```

Listing 4.4: XML Signature Reference element referencing with Id

```
<ds:Reference URI="#xpointer(/Envelope/Body)">
   <ds:Transforms>
      <ds:Transform Algorithm="http://www.w3.org/2002/06/xmldsig-
         filter2">
         <ds:XPath Filter="intersect">/child::VehiclePartLookup/
            child::Shared/child::text()</ds:XPath>
      </ds:Transform>
      <ds:Transform Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"
         />
   </ds:Transforms>
   <ds:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/
      >
   <ds:DigestValue>W6th61MXZEniXCyC8XK4LLE/+50=</ds:DigestValue>
</ds:Reference>
```

Listing 4.5: XML Signature Reference element referencing with XPath

with the specified Id attribute. XPath reference enables to use an XPath Filter 2.0 transform. For simplicity it is always transforming the `<Body>` element and it is enabled to make only one intersect XPath transform. If the Filter function is set to union or subtract, the `UnsupportedOperationException` is thrown.
The declarations of such references in XML Signature are shown in Listings 4.4 and 4.5.

4.3.5 Reference Handlers

There exist two types of references, which can be processed by the WSS Gateway: Id and XPath references. Id references are handled by the `ReferenceIDHandler` and XPath references by the `ReferenceXPathHandler`. The implementation of these modules is described below.

ReferenceIdHandler

The `ReferenceIdHandler` works in two phases (Figure 4.9). The first phase is running during the SOAP header parsing. There is not known, which elements have to be hashed for XML Signature verification. Therefore the `ReferenceIdHandler` computes digest values of all elements that can be theoretically referenced. This means, all elements containing an Id

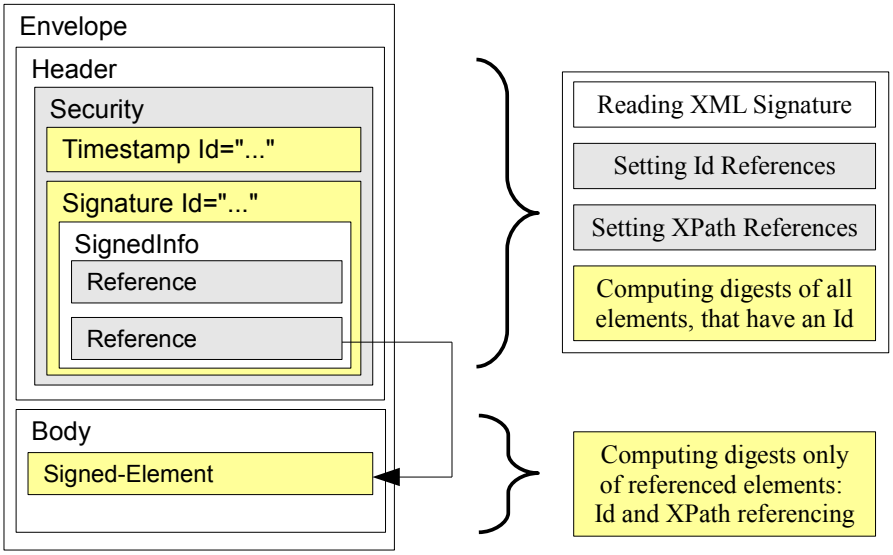

Figure 4.9: Processing references in SOAP message

attribute. For the computing it uses the default digest method (set to SHA1).
After the SOAP header ends there are already known all XML Signature properties. It is not needed to compute all references' digests. During SOAP body parsing the `ReferenceIdHandler` computes only the needed digest values.
This approach makes it possible not only to sign security tokens in security header, but also saves the memory, because only few digest values are computed.

ReferenceXPathHandler

The `ReferenceXPathHandler` processes XPath references in the body of SOAP document (Figure 4.9). It is added dynamically to the pipe at the beginning of body parsing only if some XPath reference in `ParsedDocumentProperties` exists. Otherwise this module is ignored.
For every XPath expression is responsible `SOAPXPathResultManager` (Figure 4.10). It determines for each element, if it is a *Might* match or a *True* match. A list of `SOAPXPathResultManagers` is defined in a `SOAPXPathProcessor`. This class manages them and therewith creates a connection between SPEX engine and WSS Gateway application. Before the processor starts, it allows to add new XPath expressions. Every new XPath expression is verified and rewritten into forward-only expression by `XPathRewriter` (also SPEX project component).
With the help of SPEX engine the `ReferenceXPathHandler` is informed every time a Might match or a True match element comes. But the problem is, one XPath expression can have more Might match elements. Only one of them becomes a True match, but it can happen also in the middle of Might match parsing. Therefore there is a need to compute message digest of each Might match element. It lasts not until a True match element comes (Figure 4.11). Then all the Might match elements are thrown away and only a True match element handling continues. Responsibility for this takes the class `XPathDigestResult`. At the end of True match evaluating the `XPathDigestResult` informs the `ReferenceXPathHandler` about

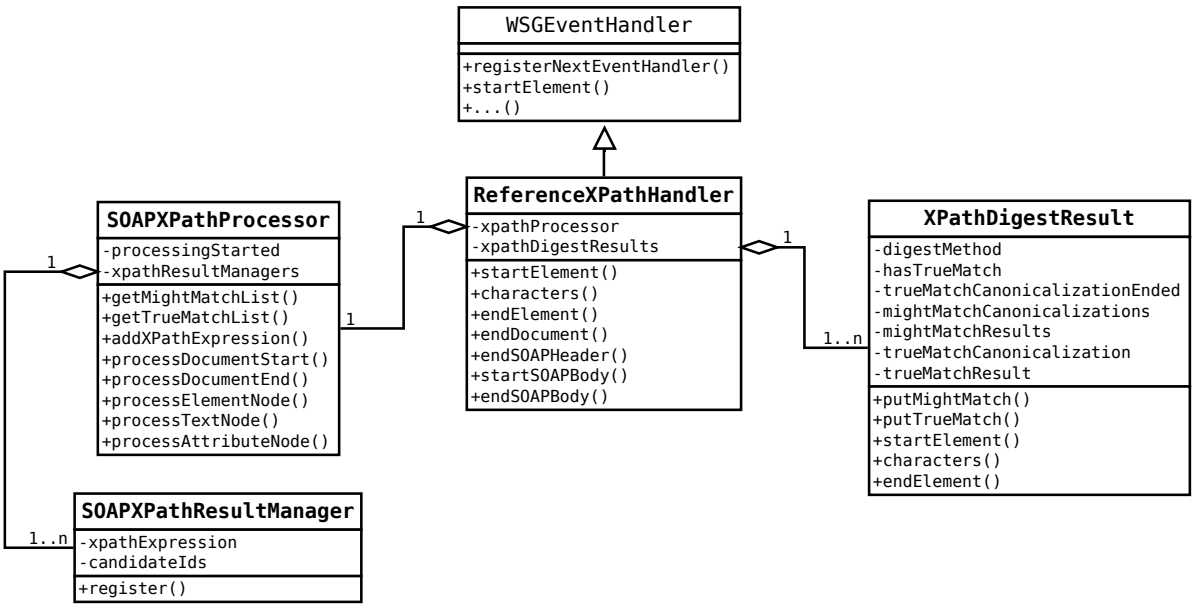

Figure 4.10: `ReferenceXPathHandler` and its connection with the SPEX engine

a new result. The result is passed to the `ParsedDocumentProperties`.
During the evaluation of this approach were found some drawbacks of SPEX implementation:

- `XpathRewriter` does not rewrite correct all the XPath expressions. An example gives a simple expression `/descendant::A[parent::B]`, that is rewritten to the expression `/descendant::B/child::A`. The correct result should be `/descendant-or-self::B/child::A`.

- SPEX can handle only expressions without namespaces. Therefore for the evaluation of XPath transformation are used only local names (processing of namespaces is ignored).

- SPEX engine has got problems with evaluation of some XPath expressions. There should not be used expressions like `/descendant::node()[...]`. They are probably too many possibilities for the expression `/descendant::node()` and the SPEX engine throws an exception. There should be used concrete nodes: `/descendant::X[...]`.

These problems reduce the functionality of `ReferenceXPathHandler` by using the above mentioned expression types. But they are not considered as a security drawback for the WSS Gateway implementation.

4.3.6 Serialization Handlers

There exist two classes responsible for the serialization of SOAP messages: `SerializationServerHandler` and `SerializationClientHandler`. Each class is used on the different WSS Gateway side. Their class diagram is depicted in Figure 4.12. The `SerializationHandler` is an abstract class. It inherits from the `WSGEventHandler` so its children can be a part of the pipeline. It owns these attributes and functions:

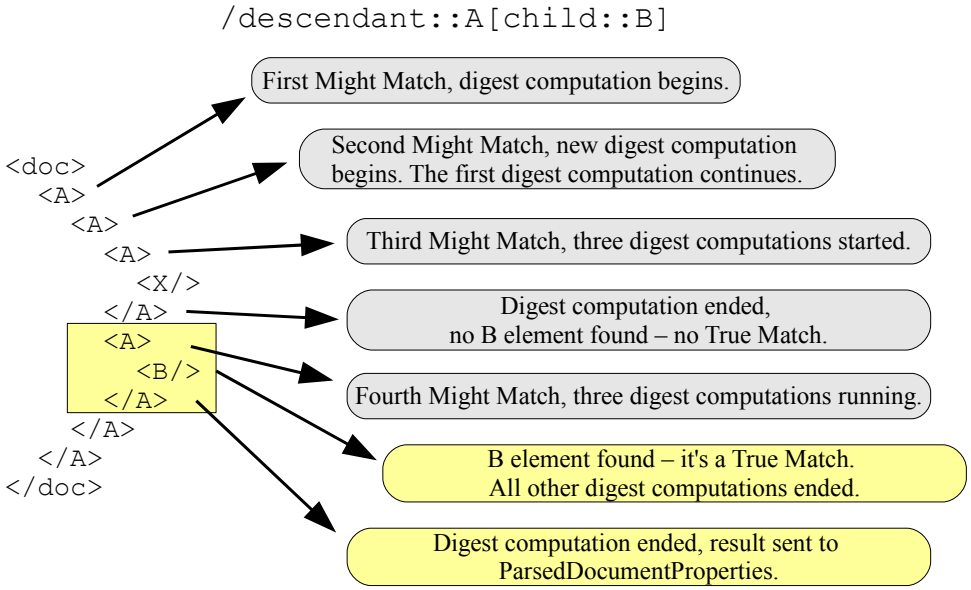

Figure 4.11: Handling XPath transformation with Might match and True match

- `writer`: An attribute handling the SOAP message writing to the output.
- `allReferencesComputed`: The boolean value indicating, if the verification has already ended.
- `elementDeclaredNamespaces`: The list of namespaces declared by an element during the parsing.
- `soapMessageBuffer`: The main buffer for SOAP message.
- `buffer`: The partial buffer for element tag creating.
- `signatureOffset`: The XML Signature offset in the `soapMessageBuffer`.
- `startPrefixMapping()`: It puts a new namespace to the `elementDeclaredNamespaces`.
- `passStartElementToBuffer()`: It takes `elementDeclaredNamespaces` and constructs a start element tag.
- `allReferencesComputed()`: If this event comes, the function `flushSOAPMessageBuffer()` is executed. Therewith is the `soapMessageBuffer` flushed to the output and cleared.

`SerializationServerHandler` extends the `SerializationHandler`. It implements basic SAX events and contains a new attribute – a collection of `signatureBuffers`. The `signatureBuffers` are used to buffer each XML Signature separately. The `SerializationServerHandler` practices these important activities during the parsing (Figure 4.13):

- It starts with buffering the whole message into the `soapMessageBuffer`.

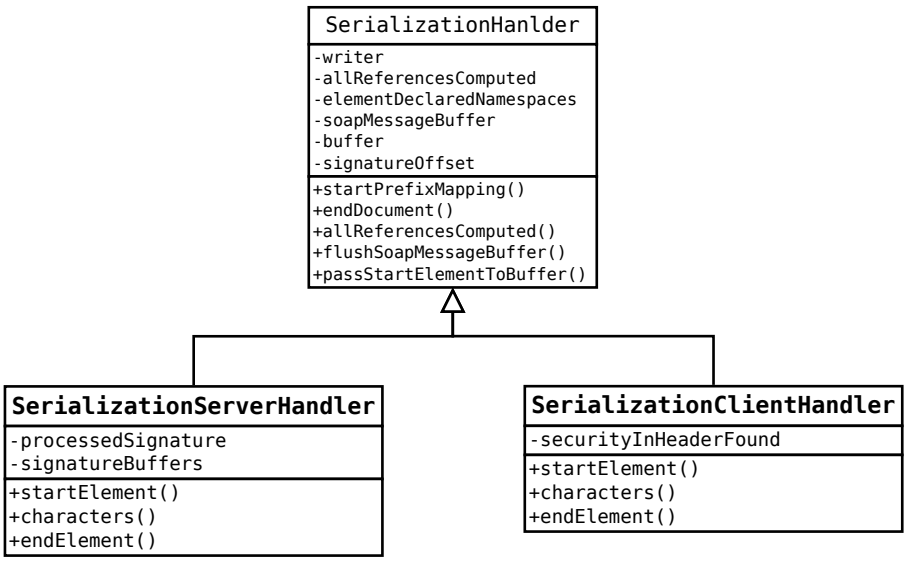

Figure 4.12: Serialization Handlers

- It stores the position of the security element in the SOAP header into the `signatureOffset`. The XML Signatures are buffered to the `signatureBuffers`.
- The message is buffered till the event `allReferencesComputed` comes. This indicates that all the elements needed for signature verification were found. If all of the references' digest values are correct, the buffer is simply flushed to the output. Otherwise (if the Gateway allows some invalid references or signatures), the incorrect signatures are placed back from the `signatureBuffers` to the `soapMessageBuffer` and flushed with this buffer together.
- The buffers are deleted and next elements are directly passed to the output.

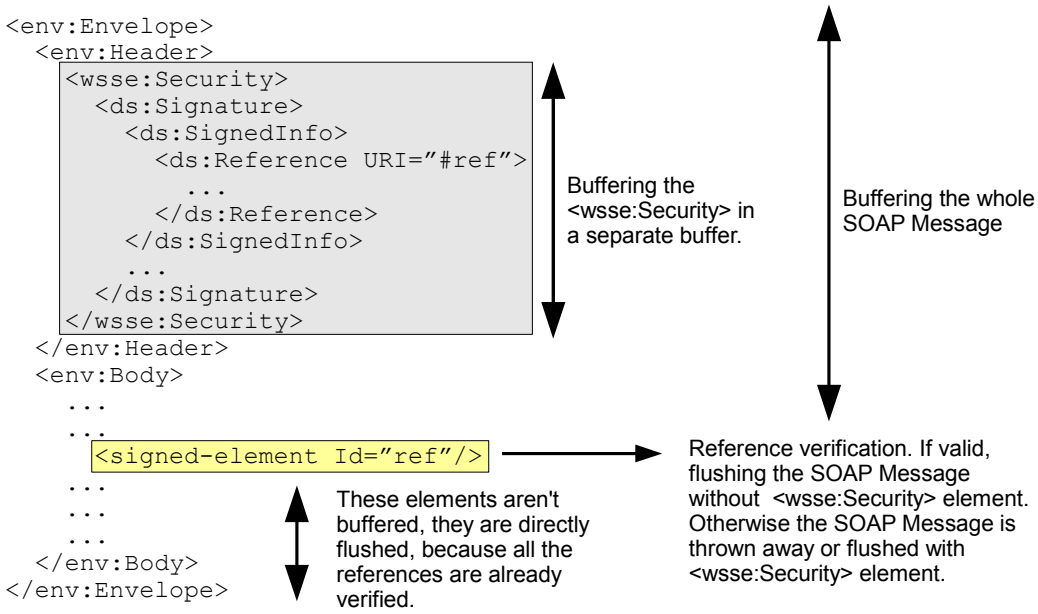

Figure 4.13: SOAP message serialization

Structure and behavior of the `SerializationClientHandler` is very similar. It also starts buffering the SOAP message and remembers the position for XML Signature. When the last referenced element ends, `SignatureBuilder` builds XML Signatures and inserts them to the `soapMessageBuffer` on the right position. Then the whole `soapMessageBuffer` is flushed to the output. Next elements are also sent directly to the output.
One can say this approach hasn't got any advantages in contrast to the DOM model processing. If the searched reference is at the end of SOAP message, almost the whole SOAP message has to be buffered. But using this approach, in memory aren't held any objects describing the XML tree structure, there is held only plain text of the SOAP message. Comparison of memory usages of both models is shown in the next Chapter.

5 Evaluation

The implemented Web Services Security Gateway (WeSSeGa) was evaluated against generated SOAP messages. Each SOAP message had a fixed header with WS-Security block containing one XML signature. The signature used RSA with SHA1 for signing and included one reference element pointing to SOAP body. The digest value was computed with the SHA1. In the first test the signed element was referenced by an Id. The WSS Gateway was getting SOAP messages with increasing size. Evaluation time and memory consumption were compared to a DOM approach. The second test used XPath referencing of signed elements. In the incoming messages was increased the number of Might match elements, which increased evaluation time and memory consumption. The last test proved the WSS Gateway's resistance against signature wrapping attacks.
The experiments ran on a machine with 64 bit AMD Sempron 3400+ (1.8 GHz) processor and 1024 MB RAM. The machine used Linux Ubuntu system (version 9.04) with Java 6.

5.1 SAX vs DOM

The WSS Gateway was first compared with the Java XML Digital Signature API [Mic06d]. This API is a standard java API for generating and validating XML Signatures. For the document processing it uses DOM. The WSS Gateway and the java API evaluated SOAP messages with signed SOAP body. The SOAP body was filled with increasing number of repeating elements. The maximum depth of the SOAP body was 3. The size of the whole SOAP message including 100,000 elements reached 5 MB.
Figure 5.1 shows the behavior of both models processing SOAP messages up to size of 1000 elements. The measured times ascend linearly during the whole processing. There can be seen only small differences between evaluation times so the usage of WeSSeGa does not bring any large improvement. The evaluation of heap memory usage in this SOAP message size area was very similar. Non of the two approaches exceeded the limit of 2 MB.
These approaches were next tested with SOAP messages including up to 150,000 elements. In each of these documents was signed the whole SOAP body. Having a look at Figure 5.2, it can be seen the heap memory consumption of the application using java Digital Signature API and the WeSSeGa. The WeSSeGa brings clearly better results. Without buffering it evaluates each SOAP message with maximum 3 MB memory consumption. If it buffers the whole SOAP message, it uses approximately three times less memory than the java Digital Signature API. The memory usage of the Digital Signature API increases linearly till the validation of SOAP messages with 70,000 elements. By the amount of 70,000 elements the growth of memory usage is slowed down and continues till the maximum – 84 MB. With this heap memory consumption the DOM approach could validate the messages with about

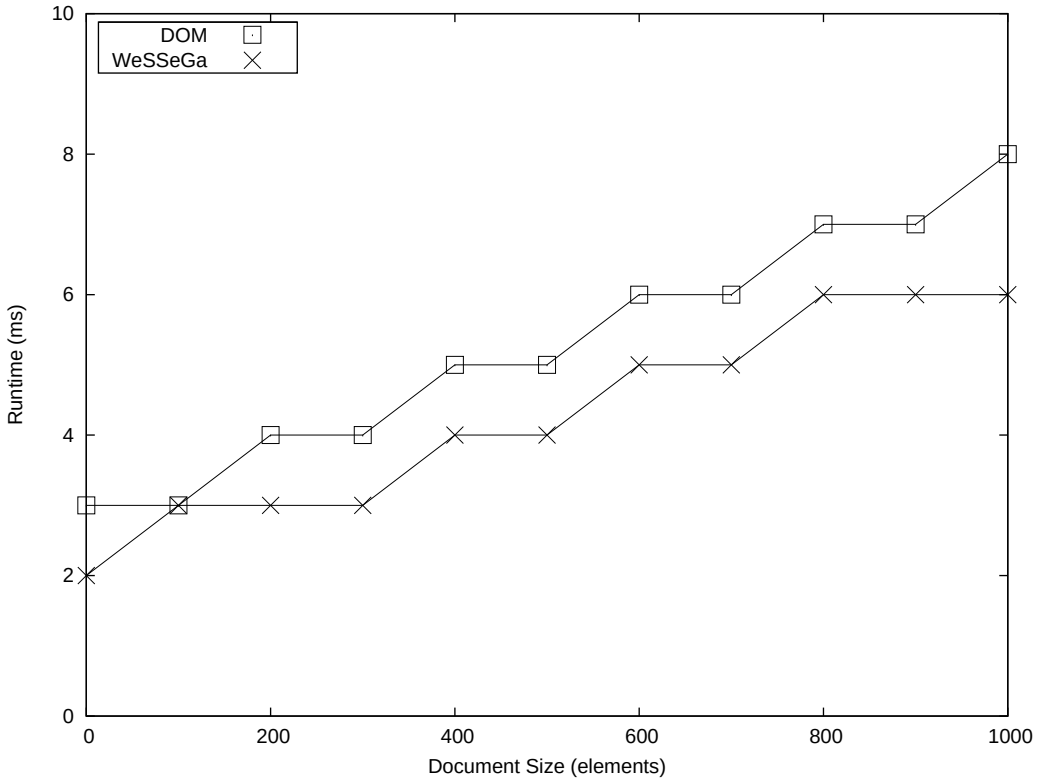

Figure 5.1: Evaluation time comparison with maximum 1000 elements

140,000 elements. Larger messages threw out-of-memory exceptions. The comparison of evaluation times is shown in Figure 5.3. As by memory consumption, the evaluation times measured with the WeSSeGa grew linearly with enlarging of the SOAP messages. The WeSSeGa could validate a SOAP message with 140,000 elements in 625 milliseconds. On the other hand the evaluation times of the DOM approach increase rapidly by the messages with more than 90,000 elements.

As the java Digital Signature API could not handle large documents the last test was made only with the WeSSeGa application. In the test were processed valid and invalid SOAP messages with maximum 600,000 elements. The signed element was inserted at the begin of the SOAP body. Figure 5.4 shows collected evaluation times where the runtime for correct messages increases linearly and the runtime for messages with invalid signatures stays constant. This test approves the great advantage of an event-based validation, which can interrupt the processing as soon as an error is found. During messages processing the memory consumption was increasing also linearly. By the messages with 600,000 it did not exceed 2 MB.

The values gained by the tests confirm the results of N. Gruschka et al., who investigated the use of similar WS-Security Gateway [GJD07].

The graphs presented in figures 5.2 and 5.3 show disadvantages of the DOM approach by processing large documents. The runtime of the DOM approach increases linearly only by the validation of SOAP messages with less than 100,000 elements. By the messages with more than 100,000 elements the runtime rapidly grows. This behavior could be caused by the processing of the java Garbage collector. The Garbage collector is forced to free the unused memory more times and more effcient than by processing small documents. This presumption is supported with the results of memory consumption measurements. The

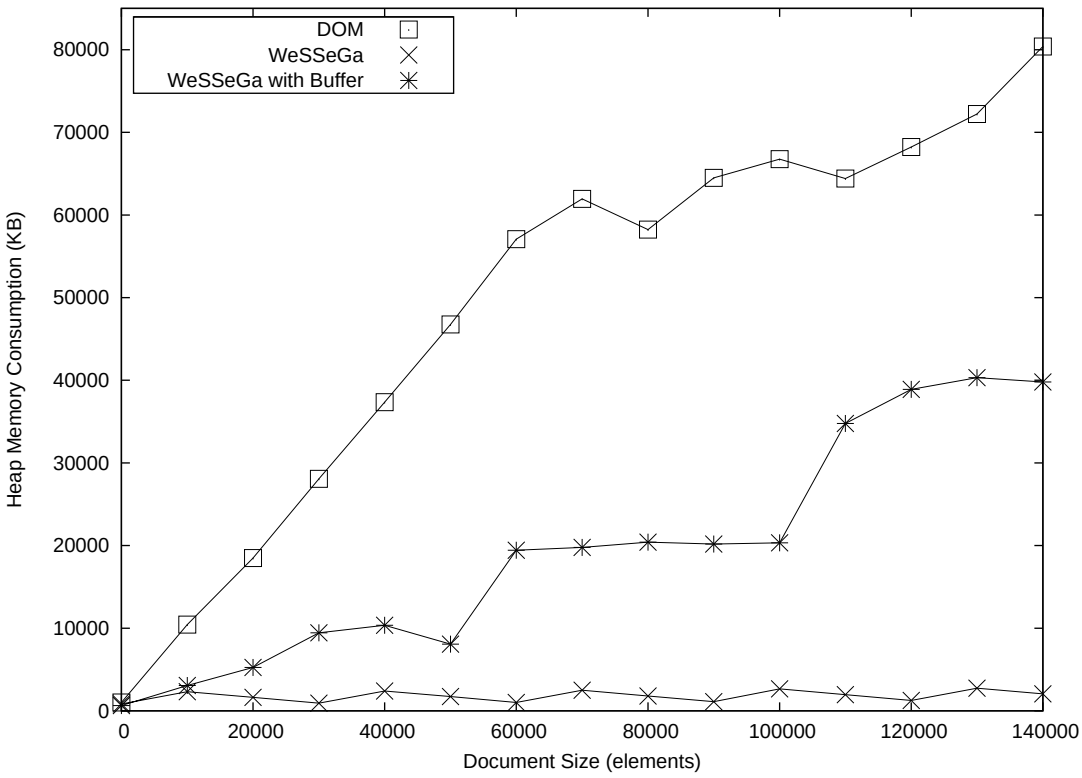

Figure 5.2: Heap memory consumption comparison with maximum 140,000 elements

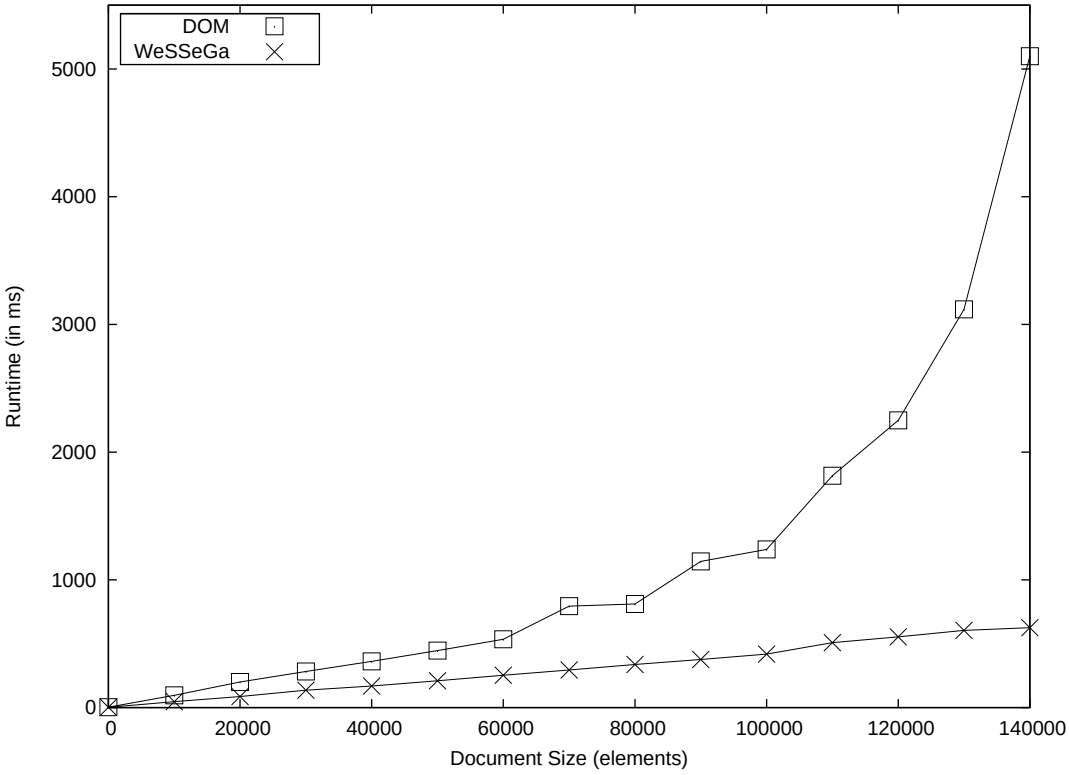

Figure 5.3: Evaluation time comparison with maximum 140,000 elements

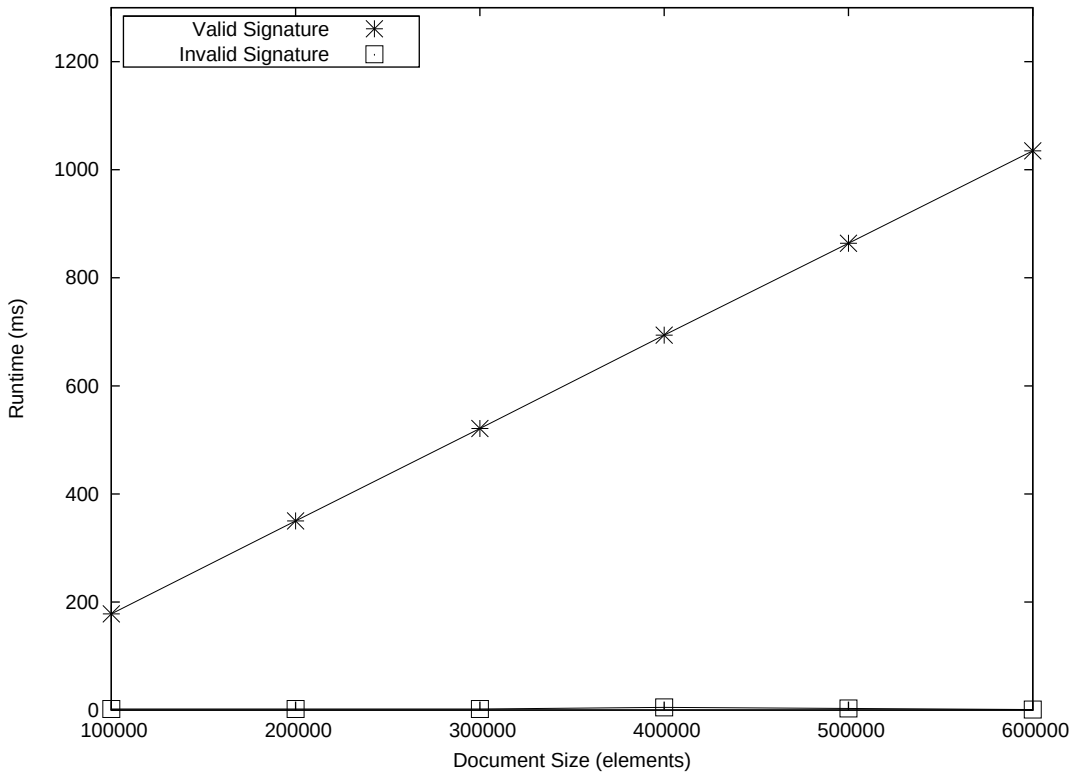

Figure 5.4: Evaluation time comparison with maximum 600,000 elements

graph shows that the memory usage increases linearly only till the processing of SOAP messages with less than 70,000. By processing larger documents the memory consumption growth is slowed down.
The graph also shows that the memory usage of the WeSSeGa application without buffer grows with increasing number of elements linearly, which is the bigest advantage in the event-based processing. On the other hand the behavior of this application with a buffer shows also some noteworthy remark. The "memory steps" in the graph are probably made by the class java.lang.StringBuilder by allocating the place for new characters in the SOAP message buffer.

5.2 Performance of XPath Referencing

The next test evaluated the performance of the used SPEX engine. In this test the WSS Gateway validated SOAP messages including XML signatures with XPath transformations. All the incoming SOAP messages contained 100,000 elements. They differed only in the number of possible Might Match elements 5.5.
The WSS Gateway evaluated SOAP messages with two XPath expressions:

- /descendant::VehiclePartSearch[child::MarketplaceDomain]: Using this XPath expression, the WSS Gateway marked each <VehiclePartSearch> element as Might Match and computed its digest value. When the <VehiclePartSearch> element ended and

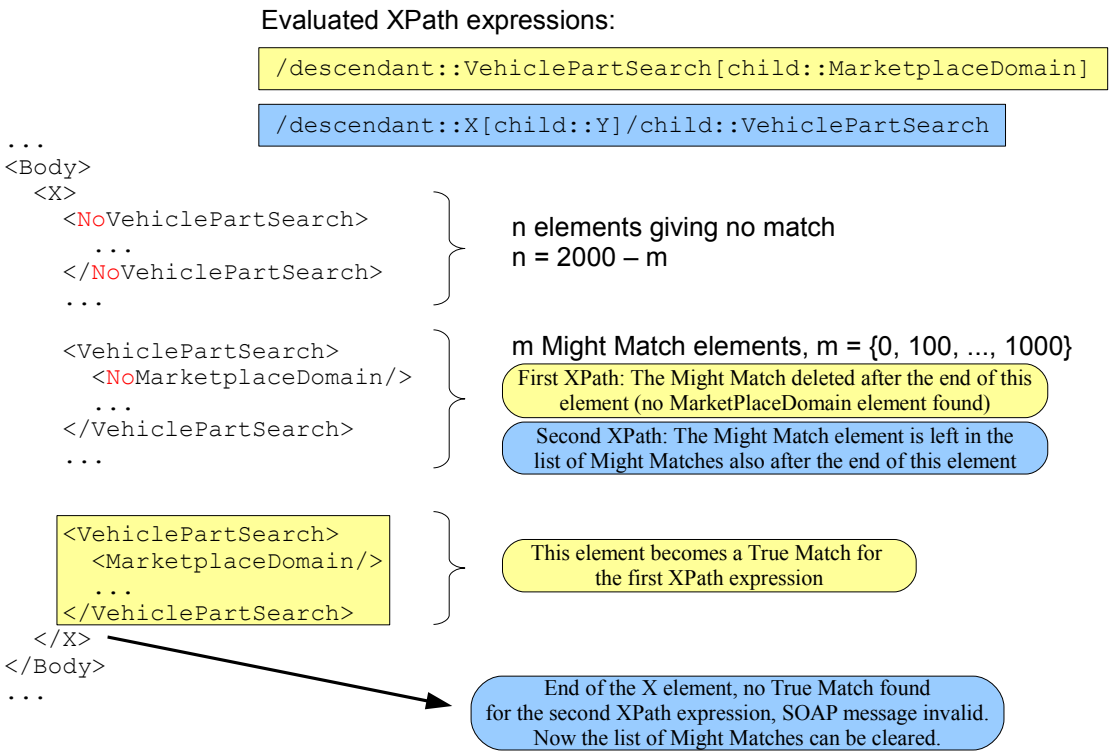

Figure 5.5: SOAP messages designed for the XPath evaluation test

it did not contain any < MarketplaceDomain>, it was deleted from the Might Match list. This way the Might Match list included during the whole parsing maximum one Might Match element.

- /descendant::X[child::Y]/child::VehiclePartSearch: Applying this XPath transformation, also each <VehiclePartSearch> element was marked as a Might Match. But when this element ended, the Might Match was not deleted from the list as the SPEX engine waited for a possible element <Y>. Therefore the list of Might Matches was not cleared until the end of <X> element.

Using these XPath expressions were achieved two different scenarios. In the first scenario was WSS Gateway deleting (closing) each Might Match directly after the Might Match ended. In the second scenario was each Might Match left (opened) in the Might Matches list till the end of the SOAP message processing.
Figures 5.6 and 5.7 present the results collected in this test. As can be seen, by increasing the number of Might Match elements by using the first XPath expression grew the runtime linearly. By the SOAP message with 1000 Might matches lied the evaluation time by about 2 seconds. The memory consumption was observed to be independent of the Might Matches number. On the other hand, when the second XPath expression was used, the memory consumption and the runtime grew rapidly with the increasing of Might Matches. By using the documents with more than 1200 Might Matches caused the validation out-of-memory exceptions.
The behavior of the WSS Gateway by evaluating SOAP messages with the second XPath expression was very similar to the behavior of the java Digital Signature API described in the previous section. The high memory consumption may have been caused by the SPEX engine, which needed more heap memory for evaluating XPath expressions with many opened

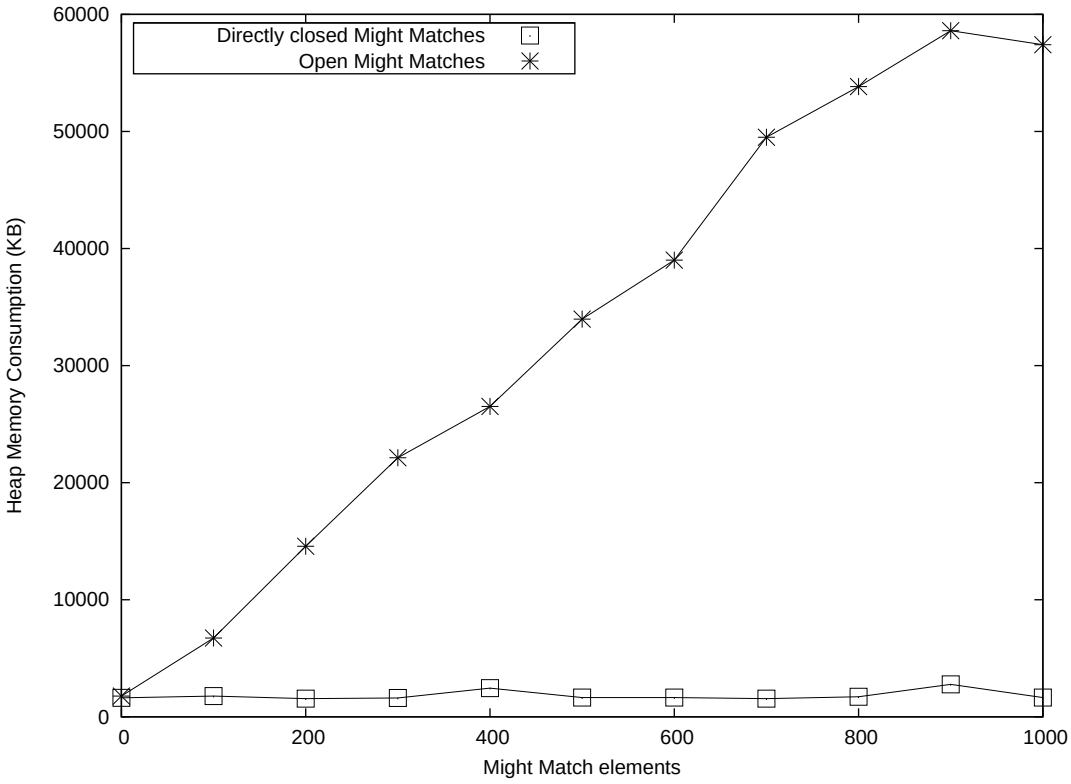

Figure 5.6: Heap memory consumption comparison by the use of XPath referencing

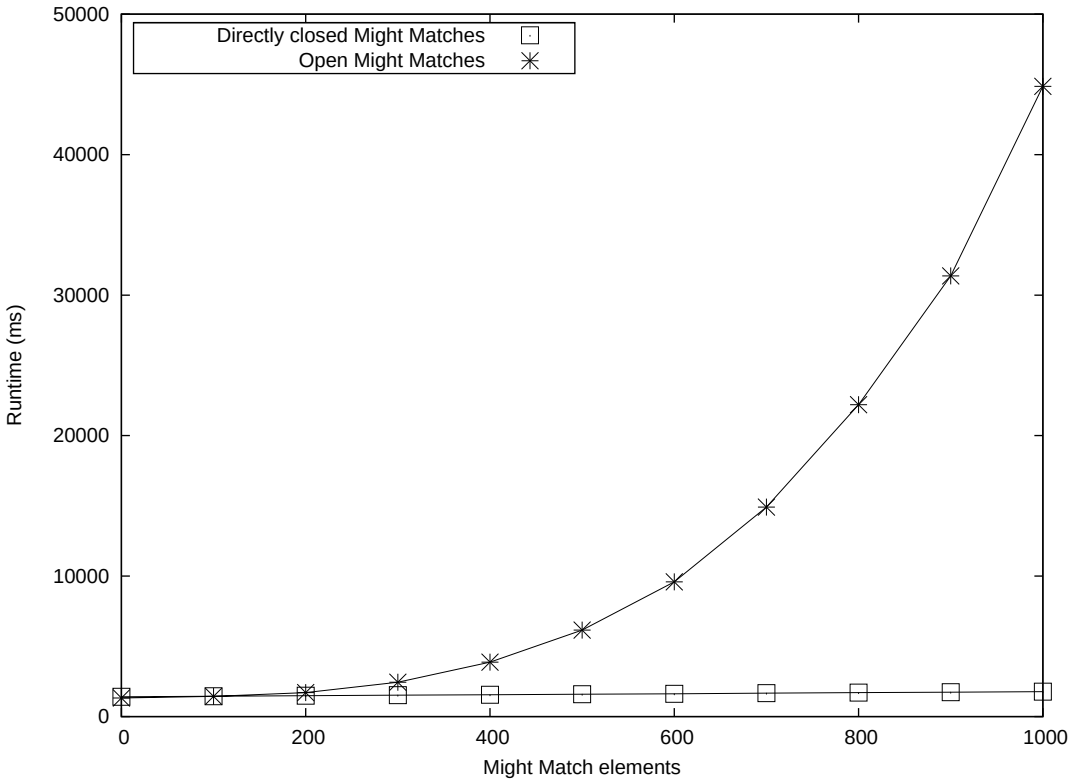

Figure 5.7: Evaluation time comparison by the use of XPath referencing

```
<soapenv:Envelope ...>
  <soapenv:Header>
    <wsse:Security ...>
      <ds:Signature>
        <ds:SignedInfo>
          ...
          <ds:Reference URI=
            "#xpointer(/soapenv:Envelope/soapenv:Body)">
            ...
          </ds:Reference>
        </ds:SignedInfo>
        ...
      </ds:Signature>
    </wsse:Security>
  </soapenv:Header>
  <soapenv:Body>
    ...
  </soapenv:Body>
  <soapenv:Body>
    ...
  </soapenv:Body>
</soapenv:Envelope>
```

The tested SOAP messages contained 2 Body elements. By the start of the second Body element the security exception was thrown.

Figure 5.8: SOAP message with two SOAP bodies

Might Match elements. Also the prototype implementation of WeSSeGa should be checked for possible improvements in the code.
The solution for this problem could bring the total restriction of XPath expressions causing Might Match elements and backward navigation. This approach was already analyzed and was named FastXPath [GJLS09]. The FastXPath uses only the expressions for forward navigation and predicates working with attributes. Such properties of XPath expressions carry directly only the searched True Match elements. Even though they enable to reference arbitrary document elements. Therefore the FastXPath is considered as well designed for event-based parsing.

5.3 Wrapping Attacks Resistance

One of the most common threats for Web Services is the signature wrapping attack. The signature wrapping attack misuses the flexibility of referencing with Ids. Executing this attack, an adversary can for instance move the signed elements from the SOAP body to an another document part and exchange them according to his needs. As the moved document block still owns the same Id, the signature is considered as valid. But the business logic executes the new function in the SOAP body.
It was approved that with using a subset of XPath expressions, the wrapping attack can be avoided [GJLS09]. Therefore the XPath transformation should also start fix at the top-most tag of the SOAP message.
In the WSS Gateway begins the XPath processing also at a fixed tag. This tag is the SOAP body. To declare the WS Gateway as resistant to wrapping attacks, it has to be shown that it is resistant to attacks, where two SOAP bodies in a SOAP message are used [GI09]. This kind of attack is based on an assumption that a DOM-based WS-Security module verifies

the second SOAP body. As the SOAP message is then validated the first SOAP body is used by a event-based business logic.
Therefore in this test SOAP messages with two SOAP bodies were constructed (Figure 5.8). The SOAP messages were sent to the WSS Gateway for the verification. As awaited, by the verification a security exception was thrown by the SOAPMessageHandler, which cares for the SOAP message consistence.
Execution of the test approved the WSS Gateway as resistant to Wrapping attacks. But this statement is built on an assumption that the evaluated SOAP messages use a set of Wrapping attacks resistant XPath expressions (e.g. FastXPath).

6 Conclusion and outlook

In this thesis, an event-based processing of secure XML documents has been discussed. The result of the discussion is the concept and implementation of an event-based Web Service Security Gateway – WeSSeGa. The WS-Security Gateway has been implemented to validate and create XML signatures in SOAP messages. Therefore it can be used on the server as well as on the client side as an external SOAP message gateway.
The presented solution has been tested against different SOAP messages. Its processing has been then compared to the standard DOM Digital Signature validation framework. The results of the tests have shown that the event-based approach brings noteworthy performance improvements considering the memory consumption and the evaluation time. The differences could be seen especially by the processing of large documents. While the implemented gateway could process the large documents without problems, the DOM validation framework has thrown out-of-memory exceptions. Therewith the suitability of the event-based approach for large SOAP messages processing has been approved.
It has been also shown that the WS-Security Gateway is able to process XPath filter transformations. The XPath filter makes the WS-Security Gateway under certain circumstances resistant against wrapping attacks. However, the results of the tests have also provided that the usage of XPath filter makes the application vulnerable against Denial-of-Service attacks. Applying the specific XPath expressions has involved the WS-Security Gateway in lengthy computations and high memory consumption.
Therefore the usage of XPath filter is one of the opened questions for the WS-Security Gateway. A solution could give the limitation of the XPath evaluator, which would process only FastXPath expressions. These expressions provide a subset of XPath expressions that are suited for fast and lightweight evaluation using an event-based approach.
Another improvement of the implementation could be developing of two new modules for XML encryption and WS-Security Policy validation. The encryption module would stand at the beginning of the pipe. It would decrypt encrypted data, send them again to the event-based parser and push the generated events to the next handlers in the pipe. The WS-Security Policy validation module would be inserted in the middle of the pipe. It would process only the events concerning the start and end of the security elements. The collected data would be next compared with the WS-Security Policy.

Bibliography

[Apa09] Apache. *Apache Commons Codec*, 2009. http://commons.apache.org/codec.

[BCD+05] François Bry, Fatih Coskun, Serap Durmaz, Tim Furche, Dan Olteanu, and Markus Spannagel. The xml stream query processor spex. *Data Engineering, International Conference on*, 0:1120–1121, 2005.

[BCD+06] François Bry, Fatih Coskun, Serap Durmaz, Tim Furche, Dan Olteanu, and Markus Spannagel. *SPEX: XPath Evaluation against XML Streams*, 2006. http://spex.sourceforge.net.

[BFH+01] Allen Brown, Barbara Fox, Satoshi Hada, Brian LaMacchia, and Hiroshi Maruyama. Soap security extensions: Digital signature. W3C recommendation, W3C, February 2001. http://www.w3.org/TR/SOAP-dsig.

[BHH+04] Steve Byrne, Arnaud Le Hors, Philippe Le Hégaret, Mike Champion, Gavin Nicol, Jonathan Robie, and Lauren Wood. Document object model (DOM) level 3 core specification. W3C recommendation, W3C, April 2004. http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407.

[Boy01] John Boyer. Canonical XML version 1.0. W3C recommendation, W3C, March 2001. http://www.w3.org/TR/2001/REC-xml-c14n-20010315.

[BTHL06] Tim Bray, Richard Tobin, Dave Hollander, and Andrew Layman. Namespaces in XML 1.0 (second edition). W3C recommendation, W3C, August 2006. http://www.w3.org/TR/2006/REC-xml-names-20060816.

[com07] Jini comunity. *Jini*, 2007. http://www.jini.org.

[con09] The Bouncy Castle contributors. *Bouncy Castle Crypto APIs*, 2009. http://www.bouncycastle.org.

[DC99] Steven DeRose and James Clark. XML path language (XPath) version 1.0. W3C recommendation, W3C, November 1999. http://www.w3.org/TR/1999/REC-xpath-19991116.

[EGV95] Ralph Johnson Erich Gamma, Richard Helm and John Vlissides. *Design Patterns: Elements of Resusable Object-Oriented Software*. Addison-Wesley Professional, 1995.

[ER02] Donald Eastlake and Joseph Reagle. XML encryption syntax and processing. W3C recommendation, W3C, December 2002. http://www.w3.org/TR/2002/

REC-xmlenc-core-20021210/.

[FF04] Eric Freeman and Elisabeth Freeman. *Head First Design Patterns*. O'Reilly, 2004.

[Fie00] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[GI09] Nils Gruschka and Luigi Lo Iacono. Vulnerable cloud: Soap message security validation revisited. *Web Services, IEEE International Conference on*, 0:625–631, 2009.

[GJD07] Nils Gruschka, Meiko Jensen, and Torben Dziuk. Event-based application of ws-security policy on soap messages. In *SWS*, pages 1–8, 2007.

[GJLS09] Sebastian Gajek, Meiko Jensen, Lijun Liao, and Jörg Schwenk. Analysis of signature wrapping attacks and countermeasures. In *ICWS '09: Proceedings of the 2009 IEEE International Conference on Web Services*, pages 575–582, Washington, DC, USA, 2009. IEEE Computer Society.

[GLH06] Nils Gruschka, Norbert Luttenberger, and Ralph Herkenhöner. Event-based soap message validation for ws-securitypolicy-enriched web services. In *SWWS*, pages 80–86, 2006.

[Gro06] Object Management Group. *CORBA Component Model Specification*, 2006. http://www.omg.org/cgi-bin/doc?formal/06-04-01.

[HBN+04] Hugo Haas, David Booth, Eric Newcomer, Mike Champion, David Orchard, Christopher Ferris, and Francis McCabe. Web services architecture. W3C note, W3C, February 2004. http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/.

[HRY09] Frederick Hirsch, Thomas Roessler, and Kelvin Yiu. Xml security algorithm cross-reference. W3C recommendation, W3C, July 2009. http://www.w3.org/TR/xmlsec-algorithms.

[JG07] Meiko Jensen and Nils Gruschka. An event pipeline pattern for stream-based xml processors. 2007.

[KL06] Chris Kaler Kelvin Lawrence. Web services security v1.1. Technical report, OASIS Open, February 2006.

[KL07] Chris Kaler Kelvin Lawrence. Ws-securitypolicy 1.2. Technical report, OASIS Open, July 2007.

[LCSG05] Wei Lu, K. Chiu, A. Slominski, and D. Gannon. A streaming validation model for soap digital signature. In *HPDC '05: Proceedings of the High Performance Distributed Computing, 2005. HPDC-14. Proceedings. 14th IEEE International Symposium*, pages 243–252, Washington, DC, USA, 2005. IEEE Computer Society.

[Lin04] P. Lindstrom. Attacking and defending web service. *A Spire Research Report*, 2004.

[MA05] Michael McIntosh and Paula Austel. Xml signature element wrapping attacks and countermeasures. In *SWS '05: Proceedings of the 2005 workshop on Secure web services*, pages 20–27, New York, NY, USA, 2005. ACM.

[Meg04] David Megginson. *SAX 2.0: The Simple API for XML. SAX project*, 2004. `http://www.saxproject.org/`.

[Mic06a] Sun Microsystems. *Java Cryptography Architecture (JCA) Reference Guide*. Sun Microsystems, Inc., 2006. `http://java.sun.com/javase/6/docs/technotes/guides/security/crypto/CryptoSpec.html`.

[Mic06b] Sun Microsystems. *Java I/O*. Sun Microsystems, Inc., 2006. `http://java.sun.com/javase/6/docs/technotes/guides/io`.

[Mic06c] Sun Microsystems. *The java web services tutorial (for Java Web Services Developer's Pack, v2.0)*. Sun Microsystems, Inc., February 2006. `http://java.sun.com/webservices/docs/2.0/tutorial/doc/index.html`.

[Mic06d] Sun Microsystems. *XML Digital Signature API*, 2006. `http://java.sun.com/javase/6/docs/technotes/guides/security/xmldsig/XMLDigitalSignature.html`.

[MM07] K. Scott Morrison Abbie Barbir Michael McIntosh, Martin Gudgin. Basic security profile version 1.0. Technical report, The Web Services-Interoperability Organization (WS-I), March 2007. `http://www.ws-i.org/Profiles/BasicSecurityProfile-1.0.html`.

[MWCR07] Jean-Jacques Moreau, Sanjiva Weerawarana, Roberto Chinnici, and Arthur Ryman. Web services description language (WSDL) version 2.0 part 1: Core language. W3C recommendation, W3C, June 2007. `http://www.w3.org/TR/2007/REC-wsdl20-20070626`.

[NHK+07] Henrik Frystyk Nielsen, Marc Hadley, Anish Karmarkar, Noah Mendelsohn, Yves Lafon, Martin Gudgin, and Jean-Jacques Moreau. SOAP version 1.2 part 1: Messaging framework (second edition). W3C recommendation, W3C, April 2007. `http://www.w3.org/TR/2007/REC-soap12-part1-20070427/`.

[OHV+07] David Orchard, Frederick Hirsch, Asir S Vedamuthu, Prasad Yendluri, Toufic Boubez, Ümit Yalçinalp, and Maryann Hondo. Web services policy 1.5 - framework. W3C recommendation, W3C, September 2007. `http://www.w3.org/TR/2007/REC-ws-policy-20070904`.

[RBH02] Joseph Reagle, John Boyer, and Merlin Hughes. XML-signature XPath filter 2.0. W3C recommendation, W3C, November 2002. `http://www.w3.org/TR/2002/REC-xmldsig-filter2-20021108/`.

[RBr02] Joseph Reagle, John Boyer, and Donald E. Eastlake 3rd. Exclusive XML canonicalization version 1.0. W3C recommendation, W3C, July 2002. `http:`

//www.w3.org/TR/2002/REC-xml-exc-c14n-20020718/.

[Riv90] Ronald L. Rivest. The md4 message digest algorithm. In Alfred Menezes and Scott A. Vanstone, editors, *CRYPTO*, volume 537 of *LNCS*, pages 303–311. Springer, 1990.

[Riv92] Ronald L. Rivest. The md5 message digest algorithm. Technical report, Apr 1992. http://www.ietf.org/rfc/rfc1321.txt.

[RSH+08] Thomas Roessler, David Solo, Frederick Hirsch, Donald Eastlake, and Joseph Reagle. XML signature syntax and processing (second edition). W3C recommendation, W3C, June 2008. http://www.w3.org/TR/2008/REC-xmldsig-core-20080610/.

[WJD+02] Norman Walsh, Ron Daniel Jr., Steven DeRose, Eve Maler, Jonathan Marsh, and Paul Grosso. XML pointer language (XPointer). WD not longer in development, W3C, August 2002. http://www.w3.org/TR/2002/WD-xptr-20020816/.